Efficient vectors with static and dynamic storage in C++

(Wydajne wektory ze statyczną i dynamiczną alokacją pamięci w C++)

Hubert Obrzut

Praca licencjacka

Promotor: dr Marek Szykuła

Uniwersytet Wrocławski
Wydział Matematyki i Informatyki
Instytut Informatyki

13 września 2021

Abstract

We propose alternative implementations of two fundamental types of vectors in `C++`: a standard dynamically allocating *vector* and a *static vector* with a fixed maximum capacity. The obtained vectors are highly efficient, customizable, and conforming to the STL's vector interface in `C++20`. They can serve as transparent replacements of standard vectors. One of the ideas is revisiting the strategy of using `mremap` Linux system calls for very efficient in-place reallocations, which is applied in previously existing `rvector`. With an improved strategy, we obtain noticeably better results, especially for *non-trivially movable* object types. In the second part of the thesis, we develop a benchmark environment dedicated to testing vectors. The experiments involve counterpart implementations, i.e., `std::vector`, `rvector`, and the vectors from Boost, Folly, and EASTL libraries. We show that our developed vectors of both types outperform all the tested ones, and can be considered as the fastest currently existing vector implementations.

W niniejszej pracy prezentujemy alternatywne implementacje dwóch podstawowych typów *wektorów* w `C++`: standardowy wektor z dynamiczną alokacją oraz *wektor statyczny* z ograniczoną maksymalną pojemnością. Obydwie implementacje są bardzo efektywne, łatwe w konfiguracji i spełniają interfejs wektora STL w standardzie `C++20`. Mogą więc służyć jako transparentne zamienniki standardowych wektorów. Jednym z pomysłów użytych w implementacji jest wykorzystanie wywołania systemowego systemu Linux `mremap` do przeprowadzenia bardzo wydajnej realokacji, na którym opiera się istniejący `rvector`. Ulepszona strategia pozwala znacznie poprawić wyniki wydajnościowe, szczególnie dla *nietrywialnie przenaszalnych* typów obiektów. W drugiej części pracy, tworzymy specjalne środowisko testowe dedykowane dla rozważanych wektorów. Przeprowadzane eksperymenty uwzględniają konkurencyjne implementacje t.j. `std::vector`, `rvector` oraz wektory należące do bibliotek Boost, Folly i EASTL. Wyniki pokazują, że oba typy wektorów są wydajniejsze od wszystkich przetestowanych odpowiedników i mogą być uznane za najefektywniejsze z obecnie istniejących implementacji wektorów.

Contents

1 Introduction **7**

1.1 Existing `static_vector` implementations 8

1.2 Existing `vector` implementations 8

1.2.1 `std::vector` 8

1.2.2 `boost::container::vector` 8

1.2.3 `eastl::vector` 9

1.2.4 `folly::fbvector` 9

1.2.5 `rvector` 9

1.2.6 Others 9

1.3 (Non-)triviality of types in `C++` 10

2 `static_vector` data structure **13**

2.1 Static storage alternatives 13

2.2 Description 14

2.3 Usage 14

2.4 `uwr::static_vector` interface 15

2.5 Implementation 16

3 `vector` data structure **19**

3.1 Description 19

3.2 `uwr::vector` interface 19

3.3 `uwr::vector` implementation 20

3.3.1 Strategy for *trivial* types 21

3.3.2 Strategy for *non-trivial* types 22
3.4 Relocatable types and type traits 24

4 Benchmark setup **27**
4.1 Vector environment 27
4.2 Tested types 28
4.3 Unit tests 28

5 Package and User Guide **31**
5.1 Prerequisites 31
5.2 Installation and tests 31

6 Experiments **33**
6.1 Vector setup 33
6.2 System setup 33
6.3 Visualization note 34
6.4 Static vector 34
6.4.1 Alternative implementations comparison 34
6.4.2 Comparison with external implementations 37
6.4.3 Turning on `push_back` optimization 42
6.4.4 `static_vector` summary 42
6.5 `vector` results 44
6.5.1 *Trivial* types 44
6.5.2 *Non-trivial* types 47
6.6 Overriding triviality 52

7 Conclusions **55**

Bibliography **57**

Chapter 1

Introduction

vector is the most commonly used data structure in `C++` with a simple purpose: provide storage for objects, allowing random access and managing their lifetime. Despite the simplicity of the concept, several different variants of vectors and implementations of those variants have been created. Some of them are expected to work for any number of stored elements, others for only a limited number, yet others expect to store a small number of elements – are optimized for this case – but still work for larger ones. Because of the heavy vector usage in virtually all `C++` software, its efficient implementation is highly desirable.

This paper concerns two variants of vectors: `static_vector` and `vector`. Both of them store objects in a contiguous memory block, which allows constant time access to the elements and is cache-friendly, especially during sequential iteration. Both of them also offer dynamic resizing. They implement a common vector interface. However, while `static_vector` – as the name suggests – uses a storage of a fixed size, `vector` uses a dynamic one – it reserves memory as needed, which is not possible in the `static_vector` case. This comes with a certain, non-zero allocation/reallocation cost. Static storage, thus high performance, makes a `static_vector` useful in scenarios where the maximum number of elements that we want to store is limited or known beforehand. On the other hand, when we do not know this maximum or do not want to keep all required memory reserved all the time, `vector` comes very handy. To allow an efficient element insertion, the `vector`'s capacity is usually greater than the number of elements that it stores. When its extra capacity is depleted, it expands the storage according to some growth policy, so that insertion at the end of the container has constant amortized time complexity.

`vector` universality makes it the most widely used data structure in all `C++` software. This is the reason why there exist multiple `vector` implementations with different allocation and growth strategies. In this thesis, we propose our implementation that uses, in particular, `mremap`[14] Linux system call that allows in-place reallocation, thus avoids high reallocation cost. Additionally, we provide `static_vector`, which is a fast alternative to the existing implementations.

1.1 Existing `static_vector` implementations

There is no `static_vector` implementation currently in STL. Having said that, `static_vector` is available in a few other popular libraries – Boost, Folly and EASTL.

`static_vector` has a pretty straightforward implementation. Because it uses static memory, there are no decisions to be made regarding allocation or deallocation - objects are part of the container itself. However it has to be noted that both `boost::static_vector` and `uwr::static_vector` are implemented as standalone types, while EASTL and Folly implement `static_vector` as a special case of their `small_vector` types – `eastl::fixed_vector` and `folly::small_vector`, respectively. `small_vector` is a container that starts with static storage, but switches to heap allocation after a certain threshold. Both EASTL and Folly allow disabling this behavior through template parameters:

```
1 template<class T, size_t C>
2 using eastl_static_vector = eastl::fixed_vector<T, C, false>;
```

Listing 1.1: `static_vector` type in EASTL

```
1 template<class T, size_t C>
2 using folly_static_vector = folly::small_vector<T, C, NoHeap>;
```

Listing 1.2: `static_vector` type in Folly library

1.2 Existing `vector` implementations

1.2.1 `std::vector`

`std::vector`[16] is the most widely used implementation of `vector` data structure, which is provided by the Standard Template Library (STL) together with a default allocator. Moreover, it is a basis for many other structures, as stacks and heaps, also provided in STL. It uses `operator new` to allocate memory and `operator` delete to deallocate. It does not use any specific operator for reallocation. Instead, it employs *allocate*, *copy*, *deallocate* scheme. Its *growth factor* is always 2.

1.2.2 `boost::container::vector`

`boost::vector`[2] is the Boost's alternative `vector` implementation. As usual, it uses `new operator` for allocations and `delete operator` for deallocations. The default *growth factor* is equal to 1.6, but Boost offers the capability to change it through template parameters.

1.2.3 eastl::vector

eastl::vector[5] is developed by Electronic Arts company, as a part of Electronic Arts Standard Template Library – an open-source library of containers and algorithms (EASTL). It is considered more suited for console or embedded environments. One of its interesting features is that it additionally has set_capacity function which frees excess storage. The standard function shrink_to_fit does not have that guarantee. Its *growth factor* always equals 2.

1.2.4 folly::fbvector

folly::FBVector[6] is a part of the Facebook's open-source library – Folly. It uses the most customized policies among all implementations that we are aware of. The first allocation has at least 64 bytes. Then, for small capacities (smaller than 4096) and a large ones (larger than $32 \cdot 4096$), FBVector uses its *growth factor* equal to 2. Otherwise (for medium sizes), it uses 1.5, to improve reusing previously allocated memory blocks (see [7]). Additionally, it tries to expand memory in-place during reallocation through the use of jemalloc's[12] xallocx call. It also provides IsRelocatable type trait, which allows overriding non-triviality of a type. This allows the use of the memcpy function instead of calling copy operator and destructor during the reallocation, thus improves performance.

1.2.5 rvector

rvector[20] is a vector on which our implementation is based upon. It uses mmap, munmap system calls for allocation and deallocation, respectively. For reallocation, mremap call is used, which has in-place capabilities. Its *growth factor* always equals 2 and the first allocation has at least 64 bytes.

1.2.6 Others

Note that work related to the idea of using mremap function to implement an efficient vector was known before, in an implementation called evector[18]. However, this work is old and the source code is not publicly available. Additionally, from time to time the idea of utilizing the in-place reallocation mechanism via realloc is being rediscovered (e.g, [11]), as the std::allocator ([17]) interface lacks of this possibility.

1.3 (Non-)triviality of types in C++

Before we dive deep into the descriptions of our vectors, we recall what is a *trivial* type in C++ language. This will be a recurring concept in our considerations, tests, and benchmarks.

The triviality of a type in C++ comes in different forms; however, most of the *trivial* traits are more or less related to each other. A type can be *trivially-constructible/destructible/copyable/moveable* (and there are a few additional traits). In general, a type if *trivial* if it has a compiler-provided or explicitly defaulted special member functions – constructors/operators/destructors. Moreover, all of its members, as well as its possible base classes, has to be *trivial* as well, and there cannot be any virtual functions. Examples of *(non-)trivial* types are:

```
/* Although all of its members are trivial
   it has default constructor (and destructor) defined,
   so it is non-trivial - both functions are
   empty but they are nonetheless defined. */
struct NT_type1 {
    NT_type() {}
    ~NT_type() {}
    int a, b;
};

/* std::string is non-trivial,
   so NT_type2 is non-trivial as well */
struct NT_type2 {
    std::string s;
    int a;
};

/* NT_type3 has defined move constructor,
   so it is non-trivial*/
struct NT_type3 {
    NT_type3(NT_type3&& other) {...}
    ...
};

/* NT_type4 has implicitly deleted default
   constructor, so it is non-trivial */
struct NT_type4 {
    NT_type4(int a) {...}
};

/* now T_type5 has explicitly defaulted
   default constructor, so it is trivial */
struct T_type5 {
    T_type5() = default;
    T_type5(int a) {...}
```

```
36 };
```

Listing 1.3: *trivial* and *non-trivial* type examples.

STL provides all kinds of type traits to check whether the type is *trivial* in one or multiple aspects. All of them have verbose and self-explanatory names, which correspond to what they really check (with a few minor exceptions, not interesting from the perspective of our considerations). For example, we have:

- `std::is_trivial<T>` - the one discussed above,
- `std::is_trivially_constructible<T>`,
- `std::is_trivially_destructible<T>`,
- `std::is_trivially_copyable<T>`,
- `std::is_trivially_moveable<T>`.

(Non-)triviality of a type is a very important concept when implementing containers for data, especially when we want to do it efficiently. The reason for that is the following: when our container allocates memory (whether it is on the heap or the stack), the memory is (usually) just an uninitialized block of bytes – it does not have any logic attached to it. The container is responsible for the lifetime of objects it stores in that memory – it has to create, destroy, copy, and move objects at, to, and from the memory it owns. To give an example – when copying and dealing with *trivial* types, the situation is simple – we can use, for instance, `std::memcpy` function to copy our raw data from one place to another. It is slightly more complex when we deal with *non-trivial* types – we have to invoke the copy operator (or the move operator) for every individually copied object. Although it may seem that this is a trivial operation (pun not intended) and does not need manual differentiation in the code – as the compiler will do the right job based on the triviality of the type – it has serious implications. They especially concern containers that can reallocate memory, because we cannot use certain functions for *non-trivial* data types – one example is `realloc` function, which treats memory that is reallocating as a block of bytes, not as an array of `C++` objects, so it will not invoke any move constructors/operators during reallocation.

Chapter 2

`static_vector` data structure

2.1 Static storage alternatives

When we want to have a fixed-sized container in `C++`, we are left with a few choices. The first and the simplest solution is to use a plain C-style array `T[]`. It works fine for *trivial* types, although it introduces minor inefficiency when working with *non-trivial* types – the default constructor of all the objects in the array is called during the array's creation. It may be desirable, but a very popular pattern used in programs is to immediately fill the array with our own, custom-created objects like in the following code snippet:

```
1 T objects[N];
2 for (int i = 0; i < N; ++i)
3     objects[i] = T(some, arguments);
```

Listing 2.1: Fill pattern using C-style array

What we wanted to do is to create objects with our custom arguments, but we ended up with the unnecessary creation of default constructed objects that are immediately replaced with new ones. Even worse, this would not be possible at all if the object type does not have a default constructor. This may not be a problem for small and "cheap" objects, but there is no real need to pay for what we do not use. Of course, we can get rid of that inefficiency by creating char-array of size `N * sizeof(T)` and explicitly constructing all the objects in-place with placement `new operator`, but we also have to remember that we have to explicitly destroy that objects at the end of the scope:

```
1 char buffer[N * sizeof(T)];
2 T* objects = (T*)buffer;
3 for (int i = 0; i < N; ++i)
4     new (&objects[i]) T(some, arguments);
5 ...
6 for (int i = 0; i < N; ++i)
```

```
7     objects[i].~T();
```

Listing 2.2: Avoiding default allocation with C-style array.

Although this solves our problem, the solution is not so pretty, and easy to introduce a bug.

Additionally working with C-style arrays can easily become cumbersome, as we have to pass its size as a separate function argument and explicitly copy them into other variables with *declare-copy* scheme, because they do not have "copy semantics". `std::array` comes with a rescue to the second problem. It is a wrapper around a C-style array that offers the benefits of being a container class, so supporting copy semantics, knowing its size, and having a few other handy member functions like `fill`, `swap` or `back`. However, to solve the first inefficiency we need another container type – `static_vector`.

2.2 Description

static vector is a dynamically resizeable container with static storage. Its capacity is a part of its type, thus is provided at compile-time via one of its template parameters. `static_vector` definition follows the pattern:

```
1 template<class T, size_t C>
2 class static_vector<T, C>;
```

Listing 2.3: `static_vector` declaration.

It implements the standard vector interface to manipulate data that is stored within the container – `insert`, `erase`, `push_back`, `pop_back`, ...

2.3 Usage

`static_vector` can be useful when:

- the number of objects we want to store or the upper limit on that number is known beforehand;
- the stored objects cannot be default constructed, so neither `std::array` nor C-style `T[]` can be used;
- dynamic allocation is not acceptable – with static storage and absence of dynamic allocations the static vector does not introduce any overhead compared to a C-style array;
- objects have to be a part of the container itself, e.g., for serialization/deserialization purposes.

Moreover, we can note that `static_vector` solves the inefficiency discussed above related to the unnecessary creation of default constructed objects – it does not create them unless we explicitly request that. To create an "array" of custom objects using `static_vector`, we can do the following:

```
1 static_vector<T, N> v;
2 for (int i = 0; i < N; ++i)
3     v.emplace_back(custom, arguments);
```

Listing 2.4: Custom created object storage using `static_vector`.

However, it has to be noted that `static_vector` keeps track (in one form or another) of the number of stored objects, which is not the case for `std::array` or `T[]`.

2.4 `uwr::static_vector` interface

`uwr::static_vector` is written for `C++20`, although it is fully compatible with `C++17`. All of its member functions are marked with `constexpr` keyword, so it can be used in `constexpr` functions as a dynamically resizable container to perform calculations at compile-time. We are not aware of any other implementation providing `constexpr` support at the moment.

`uwr::static_vector` stores its members in a properly aligned storage:

```
1 typename std::aligned_storage<sizeof(T), alignof(T)>::type m_data[C];
```

Listing 2.5: `uwr::static_vector` aligned static storage.

The vector implements the whole interface of `std::vector` for `C++20`, which also includes functions from `std` namespace: `std::swap`, `std::erase`, `std::erase_if` as well as ternary operator `operator<=>` newly introduced in `C++20`. Additionally, it has three extra methods:

- `fast_push_back` – by default, when we call `push_back` method on full vector, `std::out_of_range` exception is thrown. This is also the default behaviour for other implementations (e.g., in Boost). `fast_push_back` method assumes the vector is not full, resulting in an undefined behavior if it is not the case. In this way, we can get rid of an additional `if` statement, in this probably the most frequently used vector function.
- `fast_emplace_back` – analogous to `fast_push_back`.
- `safe_pop_back` – it is a convenience method. It checks whether the vector is empty. If it is the case, then it does nothing. Otherwise it does `pop_back`.

Both functions `fast_push_back` and `fast_emplace_back` can be very useful. This is especially visible for `static_vector`, because we should never `push_back` when

it is full as this will not increase its capacity. This is the reason why we decided to make `push_back` and `emplace_back` work the same as `fast_push_back` and `fast_emplace_back` respectively by default. To re-enable the original behaviour, thus being compatible with other implementations, one has to define a special variable with the following macro:

```
#define UWR_STATIC_VECTOR_OPTIM_ENABLE 0
#include <uwr/static_vector.hpp>
```

Listing 2.6: Turning off the `push_back` optimization.

Because all the other implementations have this additional `if` statement in both of these functions, we also do not disable it in our benchmarks in this thesis.

2.5 Implementation

We have considered two variants of `uwr::static_vector` – one *size-based* and one *pointer-based*. In *size-based* variant, we track objects that we store in the vector by having a counter (integer) of currently stored objects. In *pointer-based* variant, we store a pointer to the end of our current storage.

```
template<class T, size_t C>
class static_vector {
    ...
    size_type m_size;
    ...
};
```

Listing 2.7: *size-based* variant.

```
template<class T, size_t C>
class static_vector {
    ...
    T* m_end;
    ...
};
```

Listing 2.8: *pointer-based* variant.

size-based is probably the more natural one as we think of a container as having its size. *pointer-based* is an original approach, which can have an advantage in the absence of calculating the end of the storage. However, in exchange, we have to do an additional calculation for knowing its size. Thus, sometimes one implementation may achieve better efficiency than the other, and this issue is addressed in our benchmarks.

Moreover, *size-based* variant contains a minor optimization regarding the type of our counter. `static_vector` cannot store more objects than its capacity. As we

know the capacity at compile-time, we can use this knowledge to dispatch a minimal `size_type` type needed.

```
template<size_t C>
struct minimal_size_type {
    using type = typename std::conditional_t<
        C <= std::numeric_limits<std::uint_fast8_t>::max(),
        std::uint_fast8_t,
        typename std::conditional_t<
            C <= std::numeric_limits<std::uint_fast16_t>::max(),
            std::uint_fast16_t,
            std::conditional_t<
                C <= std::numeric_limits<std::uint_fast32_t>::max(),
                std::uint_fast32_t,
                std::uint_fast64_t
            >
        >
    >;
};
```

Listing 2.9: Automatic minimal size type dispatching at compile-time.

This approach lets us save some memory especially for small `static_vectors` and small type `T` stored in it. As we know, the size of a class in `C++` will be rounded to the size of its biggest member, so often we will not see the difference in `static_vector` size for different types of `m_size`. However, it is useful to have that optimization in mind as it can sometimes be utilized to save some memory. For example:

```
// size-based variant
assert(sizeof(static_vector<char, 5>) == 6);
```

Listing 2.10: Minimal size type dispatch makes `static_vector` in size-variant occupy only 6 bytes (instead of 16 with `size_t`)

`static_vector` has been designed also to be as cache-friendly as possible. For example, it is often the case that we want to move objects from source to destination using `move` operator, and then immediately destroy the objects at the source. If the type `T` is *trivial*, then destruction is an no-op and `move` operator transforms into a `memcpy`'s call. However, when `T` is *non-trivial*, we can either perform that in two loops – one for moving objects and the other one for destroying them, or we can move and destroy objects alternatingly in one loop. The first approach is easy to follow, especially when we are using STL methods like `std::destroy(first, last)` or `std::move(first, last, result)`. The second approach requires to differentiate *trivial* and *non-trivial* types. However, it may be beneficial, as we can avoid unnecessary cache misses, especially when the vector is large. This optimization is applied in `erase`, `insert`, and `swap` functions.

Chapter 3

`vector` data structure

3.1 Description

vector is a dynamically resizeable container with dynamic storage. vector's most frequently used method is a `push_back/emplace_back` function. It inserts a new element at the end of the storage that the vector manages. When the storage is full, it additionally has to perform a reallocation and possibly move its elements into a different memory area. To do this efficiently, a growth policy is employed. The key parameter of any vector's growth policy is the *growth factor* parameter. It is an indicator of how much vector's capacity will grow on the next reallocation. The most common growth factor value is 2.

Because vector is the most widely used `C++` data structure, being also a basis for more involved types, several different implementations have been proposed. We also develop our own, hoping it provides the best performance.

3.2 `uwr::vector` interface

Besides implementing the whole interface of `std::vector` for `C++20`, `uwr::vector` is fully compatible with `C++17`. All of its members are marked with a constexpr keyword. It also has the same three additional methods as `uwr::static_vector`:

- `fast_push_back`
- `fast_emplace_back`
- `safe_pop_back`

`fast_push_back` function assumes there is enough storage to add a new element to the `vector`, thus it does not has to execute an additional `if` statement. It is useful in

the *reserve-fill* schema, when we are sure that there the current capacity is sufficient for the next elements:

```
uwr::vector<int> v;
v.reserve(n);
for (int i = 0; i < n; ++i)
    v.fast_push_back(i);
```

Listing 3.1: `fast_push_back` usage pattern example.

Note that there is no optimization mechanism, like we had in `uwr::static_vector` case, to make `push_back` function work the same as `fast_push_back`. `vector` container by its definition is always expandable, thus we always want to perform capacity check in order to maintain `vector` functionality. That optimization was designed specifically for a static storage and non-expandable vector.

Additionally, it is possible to change the default *growth factor* value through the template parameters. `uwr::vector` uses `std::ratio` for that. There are a few predefined and most common growth rates in `uwr` namespace. The default growth factor for `uwr::vector` is 2. To use a custom one do the following:

```
// vector with custom growth rate
uwr::vector<T, std::ratio<7, 5>> v;

// vector with custom-predefined growth rate
uwr::vector<T, uwr::growth_rate_1_5> v;
uwr::vector<T, uwr::growth_rate_1_6> v;
```

Listing 3.2: Using `uwr::vector` with a custom growth rate.

3.3 `uwr::vector` implementation

`uwr::vector` is based on `rvector` [20] and can be seen as its improved version. Our vector has larger efficiency, improved implementation of most methods, applies a better reallocation strategy, has additional customization options, and fully conforms to C++20 support. Additionally, `uwr::vector` offers support of explicitly marking that a given type is trivially relocatable, even if it has the copy and/or move operators defined. By doing that we can significantly improve the performance in some cases.

The main idea applied in `uwr::vector` (as in `rvector`) is to use `mremap` Linux system call to perform reallocations of blocks of memory. In order to do that, the memory area has to be allocated using `mmap` function:

```
return (T*) mmap(NULL, n * sizeof(T),
                 PROT_READ | PROT_WRITE,
                 MAP_PRIVATE | MAP_ANONYMOUS,
```

```
4                   -1, 0);
```

Listing 3.3: `mmap` call for memory allocation

`mremap` first tries to expand the mapping in-place. It offers an optional capability, through flag arguments, to automatically move mapping to a different address in the case of failure.

```
1 T* new_addr = (T*)mremap(old_addr, old_size, new_size, flags);
2 if (new_addr == (T*)-1) {
3     // failure
4 } else {
5     // success
6 }
```

Listing 3.4: `mremap` call example.

However, this causes a problem in the case of *non-trivial* types (specifically, *non-trivially-moveable*), as the move constructor has to be called for every single moving object, and this requires both old and new blocks allocated at the same time. In this case, we have no other option but to allocate a new block using `mmap`, move the data from the old block to the newly allocated, and deallocate the old block using `munmap` call:

```
1 munmap(data, n * sizeof(T));
```

Listing 3.5: `munmap` call for memory deallocation

Because the use of `mremap` differs depending on the triviality of a type, to achieve the best results, we applied different allocation strategies for both *trivial* and *non-trivial* types.

3.3.1 Strategy for *trivial* types

`mmap` allows allocating memory in terms of single-page precision, thus we first use `malloc`[13] for smaller memory allocations (under *page size*, which is typically 4,096 nowadays), to save memory for smaller vectors and guarantee using no more than *growth factor* times more space than required. This, however, has one exception – the first allocation has at least 64 bytes. It is the same mechanism as the one used in `folly::vector` implementation. It allows saving up on some additional and unnecessary reallocations for really small vectors. Moreover, 64 bytes is the most common size of the cache line. It does not induce much memory overhead, while still helps in the most frequent vector's use patterns. After one page-size threshold, we start using memory mappings. The triviality of a type lets us to make use of `MREMAP_MAYMOVE` flag during reallocation:

```
1 return (T*)mremap(data,
2                   old_capacity * sizeof(T),
```

```
3                   new_capacity * sizeof(T),
4                   MREMAP_MAYMOVE);
```

Listing 3.6: `mremap` call for memory reallocation of *trivially-relocatable* types

This causes `mremap` to always succeed (unless the system runs out of memory) – either the mapping will be expanded in-place or it will be moved to a different place and a new address will be returned. Note that, apparently, the actual data is never physically copied from one place to another – the only change is the virtual-to-physical address mapping, thus the reallocation is very efficient, especially for big memory blocks. This can be clearly seen in *trivial* types `vector` benchmarks. Also note that the reallocation strategy for *trivial* types is the same as for `rvector`.

3.3.2 Strategy for *non-trivial* types

There are a few common things between the reallocation strategy for *trivial* and *non-trivial* types:

- the first reallocation has at least 64 bytes;
- we start by using `malloc` and switch to `mmap` after a certain threshold (this threshold will be discused later).

In the case of *non-trivial* types, we cannot use the `MREMAP_MAYMOVE` flag for `mremap` anymore. Because of that, `mremap` may fail expanding the mapping in-place, returning `ENOMEM` error.

```
1  T* new_data = (T*)mremap(data,
2                           old_capacity * sizeof(T),
3                           new_capacity * sizeof(T),
4                           0);
5
6  // on failure, allocate a new region
7  if (new_data == (T*)-1) {
8      new_data = allocate(new_capacity);
9      umove_and_destroy(new_data, data, old_capacity);
10     deallocate(data);
11 }
12
13 return new_data;
```

Listing 3.7: `mremap` call for memory rellocation of *non-trivially-relocatable* types

Experiments indicate that `mmap` and `munmap` calls together with memory copying in the case of failing in-place reallocation generate substantial time overhead. This is particularly visible when dealing with small memory blocks – the costs induced by failing reallocations outweigh the profits that result from occasional successful reallocation. In fact, `malloc` and `free` provide better performance in these cases.

This indicates that we should increase the threshold of using `mmap` beyond the size of one page (the one for *trivial* types) and use `malloc` more often. But why to not get rid of all `mmap` calls and the threshold limit altogether? Normally, `malloc` allocates memory from the heap using `sbrk`[15], which extends the data segment of the process. For allocations that exceed a certain threshold, `malloc` uses `mmap` itself. So even if we remove `mmap` completely, it still will be used silently in the `malloc` code for large allocations. Note that by using `malloc` only, we dispose of our capability to reallocate in-place – `malloc` and `realloc` do not provide any interface to check if in-place reallocation is possible. Although in-place reallocations do not improve performance when the allocations are small, they start to be of a significant cost as the vector's memory block grows.

We have choosen the `mmap` threshold for `uwr::vector` (for *non-trivial* types) to be $4 \cdot 1,024 \cdot 1,024 \cdot sizeof(long)$ instead of $4,096$ bytes. This threshold seems to be the maximum limit for `malloc` to start using `mmap` allocations. The limit was tested experimentally to give the best and most reliable performance. This hybrid strategy lets us using fast `malloc`/`free` with its `sbrk` underneath for smaller vectors, while for bigger blocks, we use `mmap`/`munmap` together with `mremap`.

The second issue is to maximize the number of successful reallocations. We have considered a few different approaches to handle the `mremap` failure. Suppose that we want to expand our capacity by a factor of 2, but `mremap` failed. Instead of immediately using the `mmap`/`munmap` pattern, we could find the maximal factor (less than 2) that yields a successful in-place reallocation. There are different approaches to achieve that:

- *Exponential*: divide our expansion factor (i.e., the growth factor minus 1) by some constant like 2 or 1.5, until `mremap` will successfully expand in-place, e.g., try to respectively grow by a factors of: 2, 1.5, 1.25, 1.125, ...
- *Linear*: linearly find the maximal growth factor over a predefined set of equally spaced numbers: 2, 1.875, 1.75, 1.625, 1.5, ...
- *Binary search*: use a binary search to find the maximal size in the range $[capacity, 2 \cdot capacity]$ rounded to the page size's precision.

We investigated that there are no real differences in terms of performance between these approaches. All of them tend to eventually find the maximal growth for a successful in-place reallocation. Even if it happens after multiple `mremap` (in *linear* and *exponent* cases) calls, it does not impose noticeable call overhead. Note that these policies are used when the vector has quite a large number of elements and managing them is a real cost here. Having said that, we decided to go with the *binary* approach, as it should be theoretically the best – it always finds the maximal growth right away, as it operates within a single-page precision. We also considered an option not to go all the way down to the single-page precision but a larger stop

threshold, but these a few additional codes mremap calls do not impose a real cost and in fact, the single-page precision search could provide a big gain (saving an execution of `mmap`/`unmap` and copying objects) in rare situations.

3.4 Relocatable types and type traits

A relocatable type in `C++` is a type that would preserve all of its invariants even if it was moved to another place in memory by a raw copy, without calling its move constructor nor move assignment operator. Note that theoretically, a relocatable type can have a move constructor defined. For example, consider `std::vector`, or even better, `uwr::vector`. It stores its size, capacity, and a pointer to the allocated storage. It does not matter where it is located in the memory. If moved to a different place, it would work correctly as before, as it is not aware, nor it should be, of the place in the memory it is located at. However, notice that this type has defined all of the move constructor, the move operator, the copy constructor, and the copy operator. So in the `C++` world, it is considered as not *trivially-movable*. There already exists a proposal to add `is_trivially_relocatable` type trait to the standard[19]. However as of now, it only exists in the experimental branch of `clang` compiler `x86-64 clang (experimental P1144)`.

Note that most of the types are intrinsically *trivially-relocatable*. The only truly *non-trivially-relocatable* types in `C++` (mentioned in [7]) are:

- Objects that store interval pointers to themselves (or one of its members). `uwr::static_vector` in the *pointer-based* variant is exactly that type.
- Objects that are observed and need to update "observers" that store pointers to them.

Often, however, it is possible to refactor such types to be *trivially-relocatable*.

A few useful examples from STL of *trivially-relocatable* types are:

- `std::vector`;
- `std::unorederd_set`, `std::unordered_map`;

and *non-trivially-relocatable* types:

- `std::string` – Small String Optimization makes it a *non-trivial* type, as it stores a pointer to itself. Bigger `strings` (of more than 15 characters) are *trivially-relocatable*, but one would have to be sure to use only big `strings`;
- `std::set`, `std::map`.

Additionally, for `uwr::vector`, we provide a whole range of type traits to define whether a type is *trivial* or not:

```
namespace uwr::mem {

/* T() */
template<class T>
inline constexpr bool is_trivially_default_constructible_v =
    std::is_trivially_default_constructible_v<T>;

/* T(const&) */
template<class T>
inline constexpr bool is_trivially_copy_constructible_v =
    std::is_trivially_copy_constructible_v<T>;

/* T(&&) */
template<class T>
inline constexpr bool is_trivially_move_constructible_v =
    std::is_trivially_move_constructible_v<T>;

/* op(const&) */
template<class T>
inline constexpr bool is_trivially_copy_assignable_v =
    std::is_trivially_copy_assignable_v<T>;

/* op(&&) */
template<class T>
inline constexpr bool is_trivially_move_assignable_v =
    std::is_trivially_move_assignable_v<T>;

/* ~T() */
template<class T>
inline constexpr bool is_trivially_destructible_v =
    std::is_trivially_destructible_v<T>;

/* T(&&) && ~T() */
template<class T>
inline constexpr bool is_trivially_relocatable_v =
    is_trivially_move_constructible_v<T> &&
    is_trivially_destructible_v<T>;

} // namespace uwr::mem
```

Listing 3.8: Custom *trivial* type traits.

These traits by default evaluate into their STL counterpart. However, we can override one or more of those traits for our custom type, without affecting the standard traits outside `uwr` namespace:

```
namespace uwr::mem {

template<>
```

```
4 inline constexpr bool is_trivially_relocatable_v<MyCustomType> = true;
5
6 }
```

Listing 3.9: Overriding *trivially-relocatable* type trait for a custom type.

The most important trait of them all is `is_trivially_relocatable_v`. It allows `uwr::vector` to use `mremap` call together with `MREMAP_MAYMOVE` flag, thus reallocation (remapping) is always successful. This provides a substantial performance gain.

Chapter 4

Benchmark setup

Benchmarks were performed using `google-benchmark` framework.

4.1 Vector environment

We developed a vector benchmark environment class, similar to the one proposed in [20]. For a given vector type and a tested type (e.g., `std::vector` and `int`), we keep track of our vectors in our environment. An *experiment* consists of a sequence of iterations, where one of the following action is performed:

- `construct` – constructs a small vector and adds it to the environment;
- `destroy` – destroys a vector and removes it from the environment;
- `push_back` – adds an element at the end of a vector;
- `pop_back` – removes random number of elements from the end of a vector (up to vector's size);
- `copy` – copies the objects from one vector to another vector using copy assignment operator;
- `erase` – erases a random range of vector contents;
- `insert` – inserts new elements into a vector (possibly in the middle).

All the actions described above are chosen with equal probability. Moreover, they are repeated multiple times within one iteration.

However, to test our containers fully and in different kinds of situations, three types of benchmark suites were designed:

- `push-only` – we perform `push_back` action only with a fixed number of vectors in the environment. This is often a use case when we use only `push_back` and

have some small, predefined number of vectors, but these vectors grow large. This benchmark is growth-focused, which is the most fundamental vector's functionality.

- `push-cons-dest` – we perform `construct`, `destroy`, and `push_back` actions. This time, the number of vectors in the environment is dynamic and can be large. Vectors are constructed, destroyed, some of them grow – this is often a recurring pattern in programs.
- `all` – we perform all the defined actions. This time vectors can grow or shrink not only one element at a time since we execute `insert` and `erase` methods. Moreover, we tend to have more vectors in the environment compared to the `push-cons-dest` case.

Although benchmarks are slightly different from each other and they may emphasize different parts of code, they tend to measure overall vectors' performance. This is not a coincidence that `push_back` action is present in all of them, as this is indisputably the most commonly used vector method. Three of them combined should give a good picture of the vector's efficiency.

4.2 Tested types

To test vectors for storing elements of different types, four such types were used in the benchmarks:

- `int` – small *trivial* type – 4 bytes;
- `std::array<int, 10>` – big *trivial* type – 40 bytes;
- `test_type` – small *non-trivial* custom type – 8 bytes; it has defined constructors, operators, and a destructor, and performs some small calculations and correctness checking inside; it should be "lighter" than `std::string`;
- `std::string` – big *non-trivial* type – 32 bytes; however, we use empty strings, so they do not allocate own memory blocks.

Of course, the exact sizes of these objects may depend on the architecture, and we specified the sizes commonly occurring in current systems. These four types represent combinations of small/big and trivial/non-trivial characteristics.

4.3 Unit tests

Additionally to the performance tests, correctness tests were written using `gtest`[10] framework. Every vector method in every variant is carefully tested to get rid of all

the bugs in the implementation, as well as be able to modify, improve, and add new implementations in the future.

Chapter 5

Package and User Guide

The vector implementation together with unit tests and benchmarks presented in this paper is available in the public repository under address:

- `https://github.com/nan0S/fast-vectors`

5.1 Prerequisites

The vector implementation itself does not have any additional external dependencies. The only one is to use:

- `gcc` compiler under Linux system
- `git` and `make` tools to follow the tutorial below

Additionally to build all the benchmarks and tests, the following dependencies are necessary:

- `google-benchmark`[9] and `gtest`[10] frameworks
- Boost[1], EASTL[4] and Folly[8] libraries
- `jemalloc`[12]

5.2 Installation and tests

To install the code locally on a machine run the following commands:

```
1 git clone https://github.com/nan0S/fast-vectors
2 make install
```

Listing 5.1: Installing the library locally.

After successful installation, we can include the code in programs with the following `#include` statements:

```
#include <uwr/container/vector.hpp>
#include <uwr/container/static_vector.hpp>
```

Listing 5.2: Including vectors in a custom program.

It is also possible to manually copy `include` folder, where all the implementation headers are contained, directly to the project directory tree and skip the installation.

There are multiple benchmarks and tests available. To run one of them, `make` command has to be used together with the benchmark/test name:

```
# benchmarks
make run-vector_benchmark
make run-static_vector_benchmark

# unit tests
make-run-vector_test
make-run-static_vector_test
```

Listing 5.3: Running tests and benchmarks locally.

Chapter 6

Experiments

6.1 Vector setup

The versions of the used libraries are:

- Boost 1.76.0-1
- EASTL 3.12.01-1
- Folly 2021.08.23

All of the tested vectors have their *growth factor* equal to 2 (except `folly::vector`, which has the *growth factor* equal to 1.5, but only for medium-sized vectors; furthermore, it is not customizable, being a part of this vector's policy). We changed the `boost::vector` *growth factor* from 1.6 (default) to 2. This change makes `boost::vector` faster (fewer reallocations) and allows a more fair comparison, apart from differences caused by different values of default parameters.

6.2 System setup

We perform the benchmarks under the following system setup:

- `CPU`: Intel Core i5-5300U CPU @ 2.30GHz, 2 cores
- `L1/L2/L3 cache`: 32KiB/256KiB/3MiB
- `RAM`: 8GiB
- `System`: Arch Linux, Linux 5.13.12-arch1-1
- `Compiler`: `g++`, version 11.1.0
- `C++ standard`: `-std=c++17`/`-std=c++20`

- `Optimization: -Ofast -flto -march=native -DNDEBUG`

6.3 Visualization note

The results are presented in the form of plots. In all the cases, the obtained values are averaged over multiple runs. Additionally, small error bars are marked on the plots to indicate 95% confidence interval.

6.4 Static vector

6.4.1 Alternative implementations comparison

In order to settle with one `static_vector` variant for further tests with external implementations, we first compare two alternative implementations described earlier – one *size-based* and one *pointer-based*, named `uwr::static_vector_size` and `uwr::static_vector_ptr`, respectively. Figures 6.1, 6.2, 6.3 show benchmark results for the three types of benchmarks described earlier. Each static vector in the benchmark has the capacity of $500,000$ elements.

There is no clear winner. Both variants seem very similar at the first glance, so unfortunately no strong preference can be formed. One could go with one implementation or the other and probably not notice any significant performance difference. However, there are some reasons to choose *size-based* variant. We will try to present them.

First of all, *size-based* variant seems to be slightly faster than *pointer-based* in the average case, which can be seen in `push-cons-dest` and `all` benchmarks – Figures 6.2 and 6.3, respectively. It was investigated that `destroy` action seems to make that small difference between the variants. This action, besides destroying the vectors, does one more thing – swaps about to be destroyed vector with some other vector from the environment. In *size-based* case, besides swapping the contents of vectors, we can just swap their sizes. However, in the *pointer-based* case, we cannot just swap their pointers, because they point to each vector's local storage, and those are located under different memory addresses – in turn, we have to do a few more calculations – compute offsets from the beginning of vector storage and calculate new pointers based on the swapped offsets. Here is a code snippet, showing this in more detail:

```
1 void swap(static_vector_size& x, static_vector_size& y) {
2     // swap actual contents of vectors
3     // ...
4     std::swap(x.m_size, y.m_size);
```

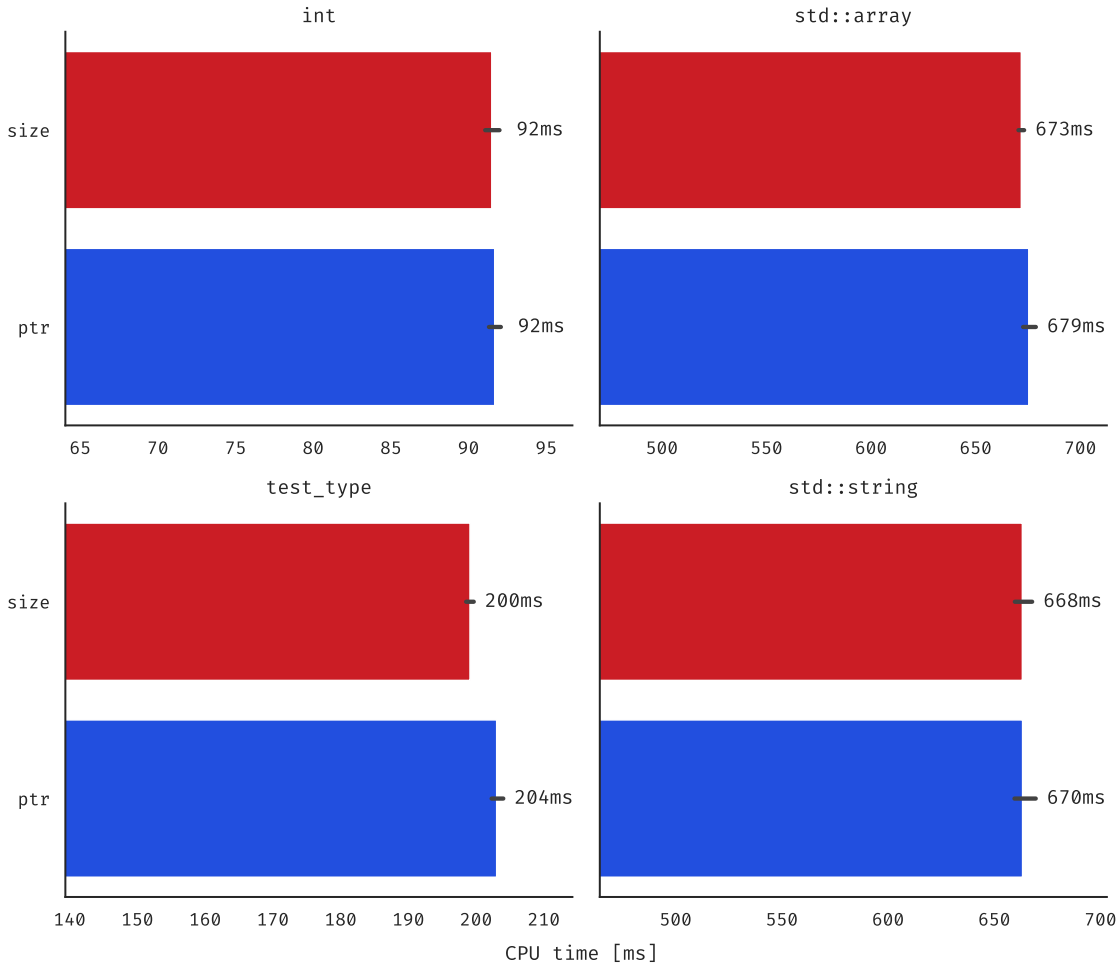

Figure 6.1: `push-only` benchmark for `int`, `std::array<int, 10>`, `test_type` and `std::string` types, comparing two alternative `static_vector` implementations.

```
}
```

Listing 6.1: `swap` implementation for *sized-based* variant

```
void swap(static_vector_ptr& x, static_vector_ptr& y) {
    // swap actual contents of vectors
    // ...
    size_type x_len = x.end() - x.begin();
    size_type y_len = y.end() - y.begin();
    x.set_end(x.begin() + y_len);
    y.set_end(y.begin() + x_len);
}
```

Listing 6.2: `swap` implementation for *pointer-based* variant

Indeed, Figure 6.4, which compares two variants in `swap` function benchmark, shows a slight advantage of *size-based* variant. Also note that `swap` function for `static_vector` is not that cheap as for regular `vector`. The data is a part of the vector object itself, so we have to swap ranges of objects, which are linear in the size of the container.

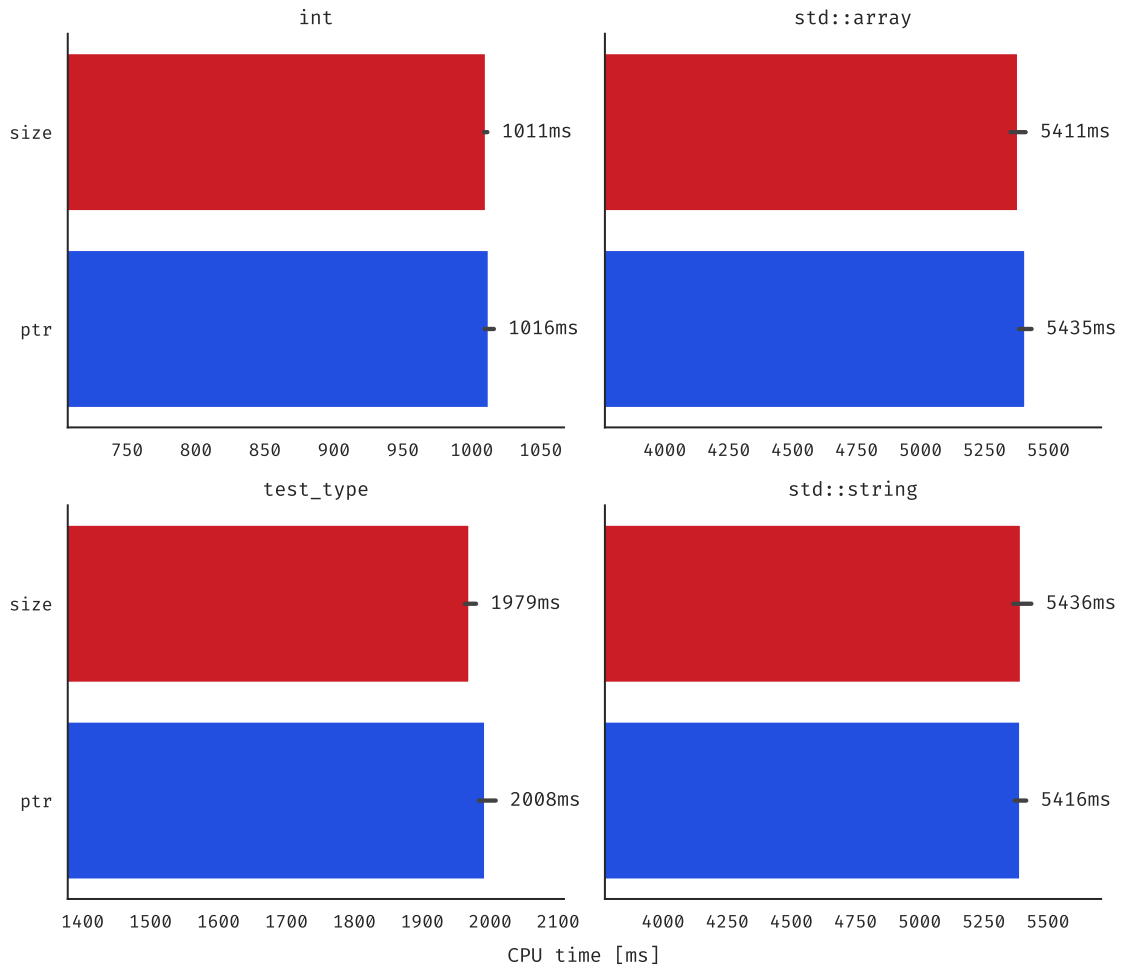

Figure 6.2: `push-cons-dest` benchmark for `int`, `std::array<int, 10>`, `test_type` and `std::string` types, comparing two alternative `static_vector` implementations.

In the usual `vector` case, it is enough to swap the internal pointers. Additionally, *size-based* variant has an advantage of being smaller in size, for most of the cases, than *pointer-based* variant – storing the size usually takes up 4 bytes of memory (with minimal size dispatching described earlier, this could be even less), whereas a pointer always takes 8 bytes on modern architectures. This could especially make a difference when we have a large number of small static vectors. Another reason to keep track of the vector's size instead of a pointer to the end of storage is that it is usually the case that we compute and want to know the size of our vector. This point of course depends on the usage, but *pointer-based* representation requires an additional computation to know its size. On the other hand, one could also say that the pointer to the end is more useful when performing `push_back` – we insert a new object at the end of the storage, thus it is required to compute that pointer in *size-based* case. However, that code is usually optimized by the compiler, especially when we do a few `push_back`s in a row. But more importantly, we surprisingly do not see that cost in the benchmark results, even in `push-only` case (Figure 6.1),

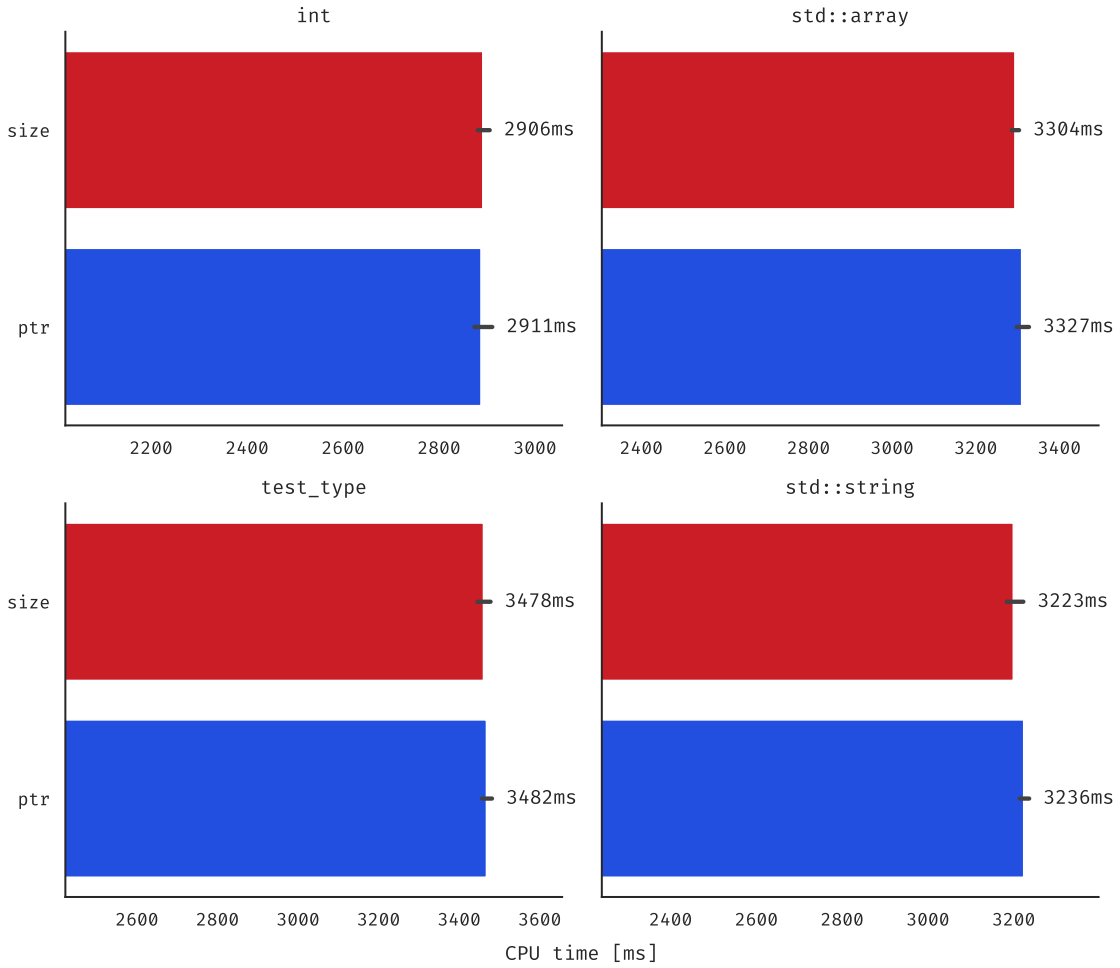

Figure 6.3: `all` benchmark for `int`, `std::array<int, 10>`, `test_type` and `std::string` types, comparing two alternative `static_vector` implementations.

which focuses on `push_back` function the most. Lastly, the *pointer-based* representation makes `static_vector` truly *non-relocatable* type. Keeping a pointer to an object's static storage forbids us from moving that object to a different memory address (without calling an additional operator, e.g., with `memcpy` function). This could yield high costs in terms of the performance when static vectors need to be relocated (e.g., being elements in `uwr::vector`), and thus should be avoided when possible.

All of these reasons makes us choose *size-based* variant as a final implementation for `static_vector`. From now on, `uwr::static_vector` represents the *size-based* option.

6.4.2 Comparison with external implementations

We perform our three main types of benchmarks – `push-only`, `push-cons-dest`, and `all`. The results are shown in Figures 6.5, 6.6, 6.7 respectively.

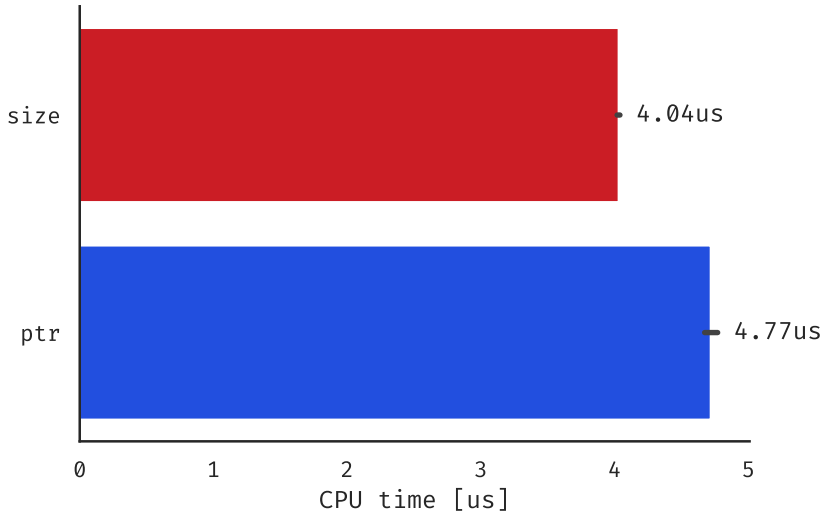

Figure 6.4: `swap` function benchmark for `std::array<int, 10>` value type, comparing alternative `static_vector` implementations.

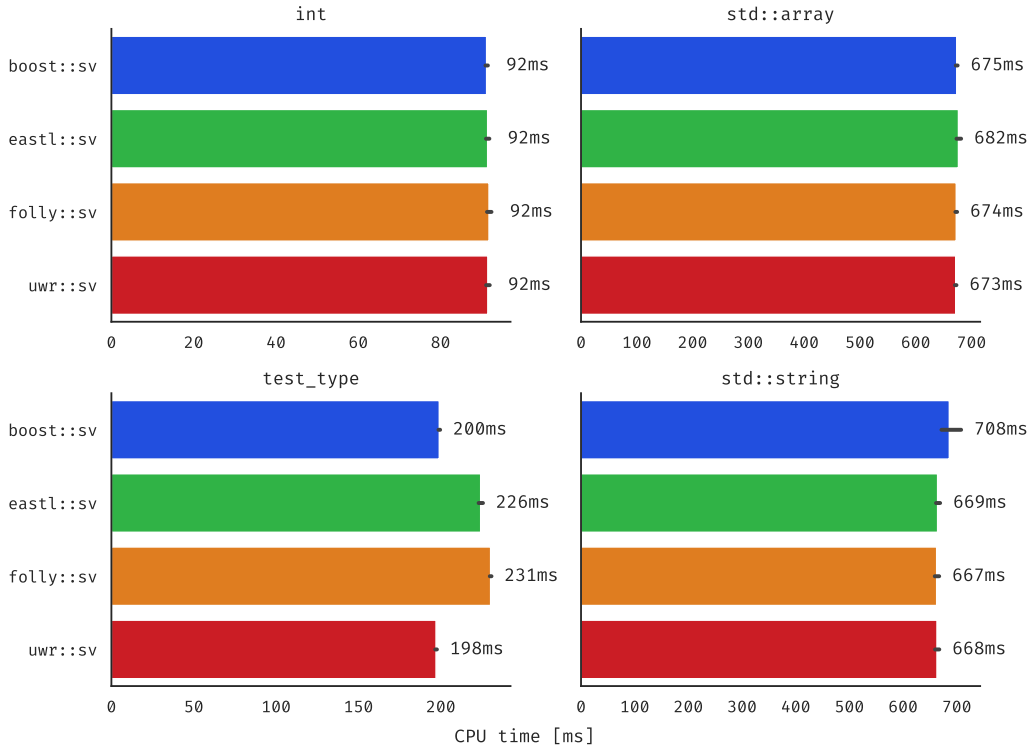

Figure 6.5: `push-only` benchmark for `int`, `std::array<int, 10>`, `test_type` and `std::string` types, comparing multiple `static_vector` implementations.

The results indicate that implementations of `static_vector`, which are special cases of `small_vector` – Folly and EASTL – tend to have worse performance. This is most likely because they are not highly optimized for this specific use case, as the code for `small_vector` should handle heap allocations as well. On the other

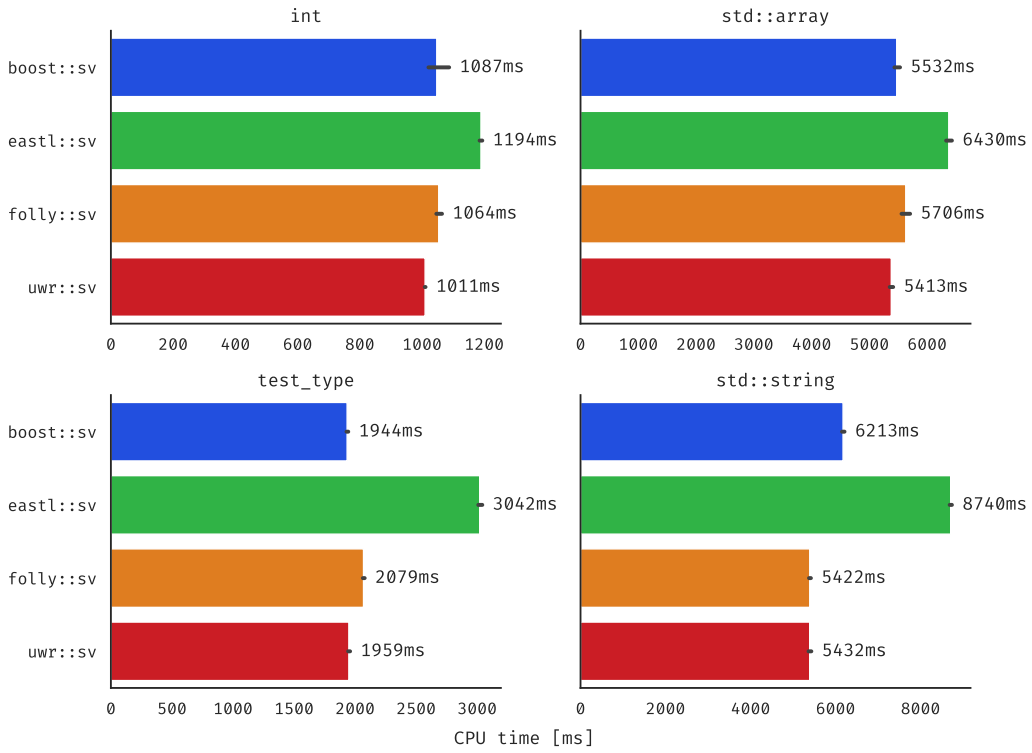

Figure 6.6: `push-cons-dest` benchmark for `int`, `std::array<int, 10>`, `test_type` and `std::string` types, comparing multiple `static_vector` implementations.

hand both `uwr::static_vector` and `boost::static_vector` perform similarly in most of the benchmarks. They are similar in a lot of ways – both implement `static_vector` as a separate type, so they are optimized for that case, both do intrinsically the same things – insert objects into a static-storage container and nothing else (in contrary to `small_vector`) – they do not allocate, deallocate, nor reallocate. However, in some cases, `uwr::static_vector` has a winning edge over `boost::static_vector`. This is most noticeable for the big type benchmarks – see `push-cons-dest` benchmark for `std::string`, where `uwr::static_vector` is 15% faster than `boost::static_vector`, and for `std::array` type, where it is 2% faster (both are shown in Figure 6.6). We can see a small speed-up in the `all` benchmark as well – not so significant, but noticeable and repeatable (Figure 6.7).

It was already mentioned that `uwr::static_vector` tries to be cache-friendly, thus, when it is possible, does one pass over the range of objects instead of two – e.g., we use:

```
1 move_and_destroy(dest, first, last);
```

instead of:

```
1 move(dest, first, last);
```

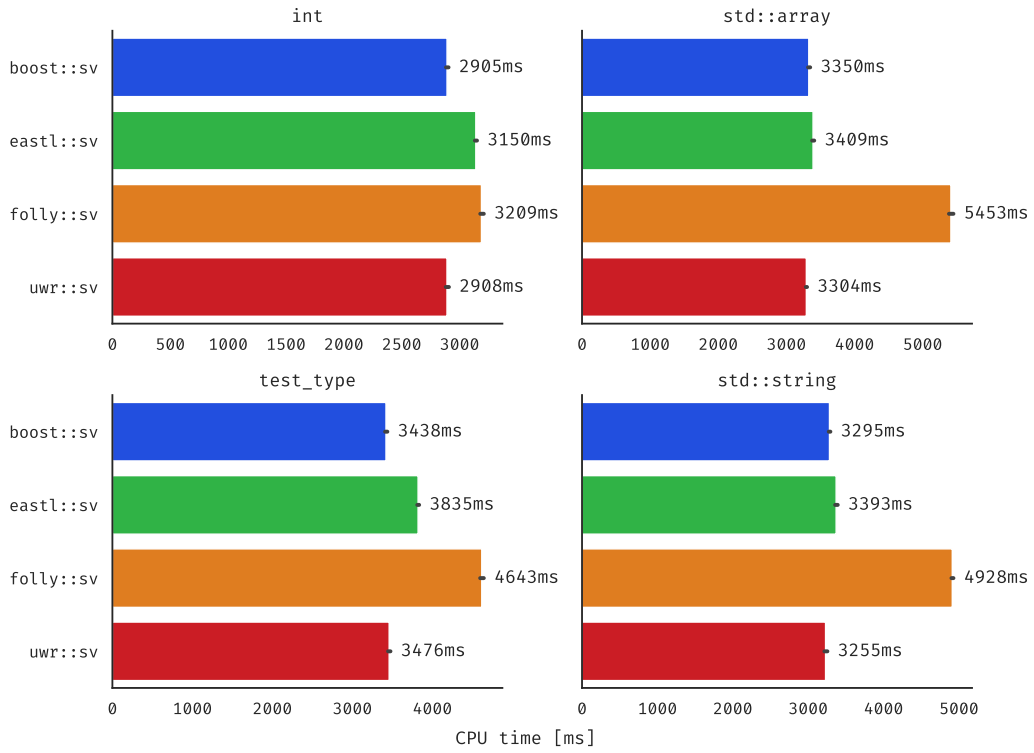

Figure 6.7: `all` benchmark for `int`, `std::array<int, 10>`, `test_type` and `std::string` types, comparing multiple `static_vector` implementations.

```
2 destroy(first, last);
```

where `move_and_destroy` function is our own custom function, while both `move` and `destroy` can be found in STL. It was investigated that `boost::static_vector` does not do that, hence tends to have worse performance. Figure 6.8 shows the results of the `insert` function benchmark for value type `std::string`, where our cache optimization was applied; cache miss ratio is also showed for that benchmark. This accounts for why `boost::static_vector` performs worse. We can see 5% speedup of our implementation compared to the Boost alternative. At the same time L1 cache miss ratio decreased from 15% to 14% – it is important to note that cache performance is so crucial, that even such a small miss rate decrease results in an observable difference in performance. This is especially true for caches on the first levels, like L1 or L2 cache. However, cache-friendliness explains the speed-up only for *non-trivial* value types like `std::string`. For *trivial* types, `destroy` function is a no-op, so using `move_and_destroy` is the same as using both `move` and `destroy` functions – `destroy` function will just be optimized away. There is another difference between our and Boost implementation in *trivial* case.

When the value type is *trivial*, like `std::array`, then `swap` function, which is used both in `push-cons-dest` and `all` benchmarks (in `destroy` action), is slower for `boost::static_vector` than for `uwr::static_vector`. The Boost's version of

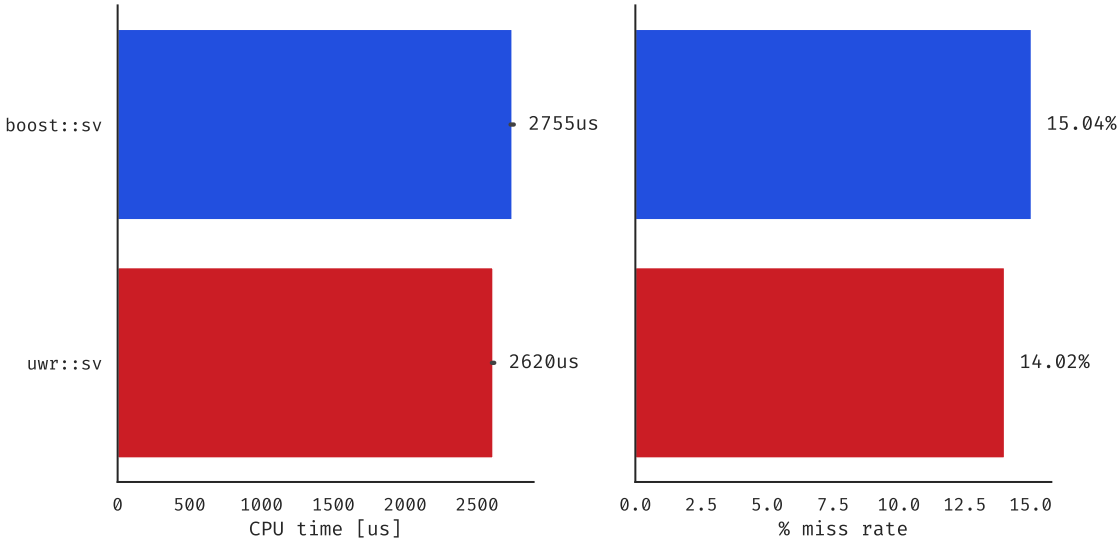

Figure 6.8: `insert` function benchmark for *non-trivial* value type – `std::string` – comparing cache optimization of `uwr::static_vector` with `boost::static_vector`. Plot on the left shows benchmark results, plot on the right shows cache miss ratio for those benchmarks.

`swap` uses loop unrolling using *Duff's device*[3] together with a small temporary byte storage, to swap two ranges of *trivial* objects. This seems to be a pretty clever construction, but it does not impose performance improvements. Quite the contrary, in our tests it degrades the performance (see Figure 6.9) compared to the normal swap using `std::swap_ranges` function, which is used in `uwr::static_vector` implementation.

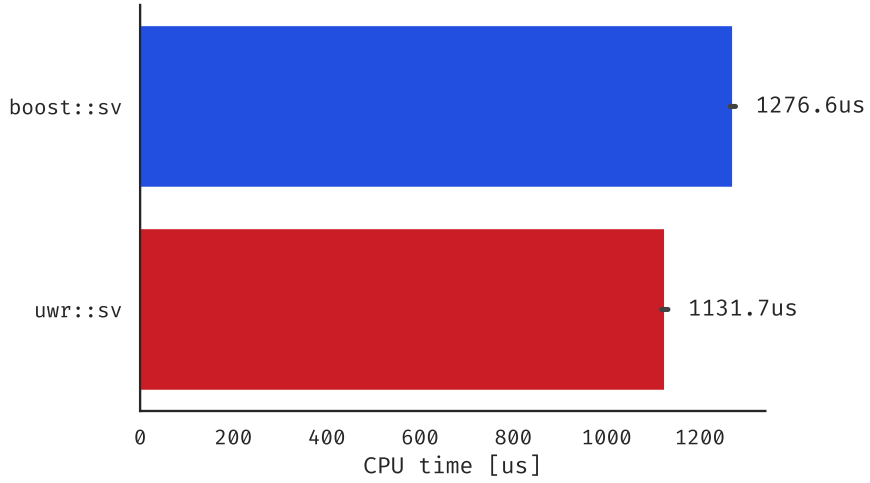

Figure 6.9: `swap` function benchmark for *trivial* value type – `std::array<int, 10>` – comparing `uwr::static_vector`'s `swap` function with `boost::static_vector`'s implementation.

6.4.3 Turning on `push_back` optimization

Finally, we enable an additional optimization turning off the `if` statement in `push_back` function, responsible for checking if the vector is full – and throwing an exception if it is. It was disabled in the previous experiments. The results are presented in Figures 6.10, 6.11 and 6.12. They only show benchmarks with `int` as the value type. For other types, there was a smaller or no difference compared to the previous results. Since `int` type is the lightest of them all, disabling one `if` statement within frequently used function impose great performance improvements. On the other hand, getting rid of `if` statement for bigger or more costly types does not improve performance that much, as other factors are more dominant.

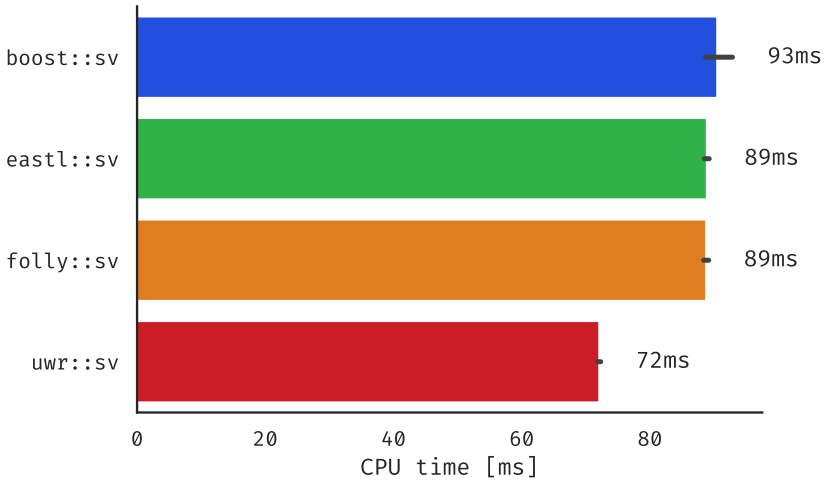

Figure 6.10: `push-only` benchmark for `int` value type, comparing `uwr::static_vector` with `push_back` optimization with alternative `static_vector` implementations.

The performance improvement in the case of `int` is substantial. We can observe $20 - 40\%$ speedup compared to the previous results. Hence, it is very beneficial to have that optimization turned on.

6.4.4 `static_vector` summary

Existing `static_vector` implementations are of two types – EASTL and Folly libraries provide `static_vector` functionality as a special-case type of its `small_vector` implementation. On the other hand, Boost together with our custom `uwr::static_vector` implement it as a separate type. The first type implementations tend to have significantly lower performance compared to those of the second type, so if only `static_vector` functionality is needed, the second type implementations should be preferred.

Our custom implementation of `static_vector` – `uwr::static_vector` – deliv-

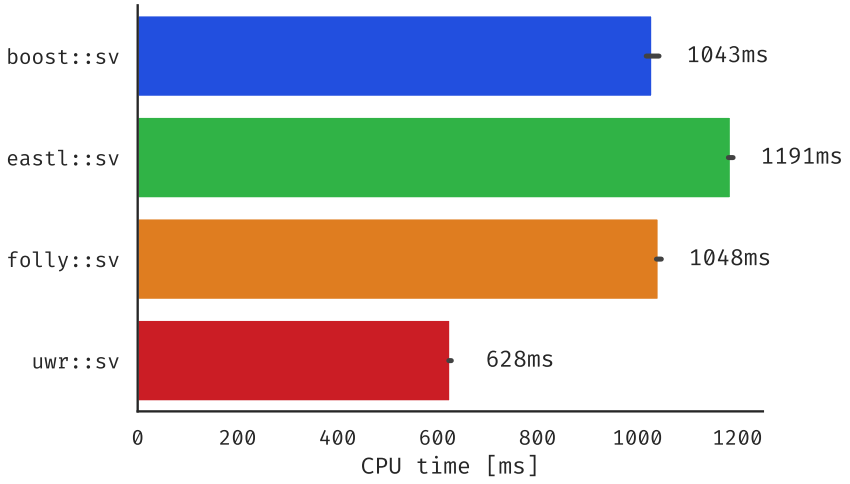

Figure 6.11: `push-cons-dest` benchmark for `int` value type, comparing `uwr::static_vector` with `push_back` optimization with alternative `static_vector` implementations.

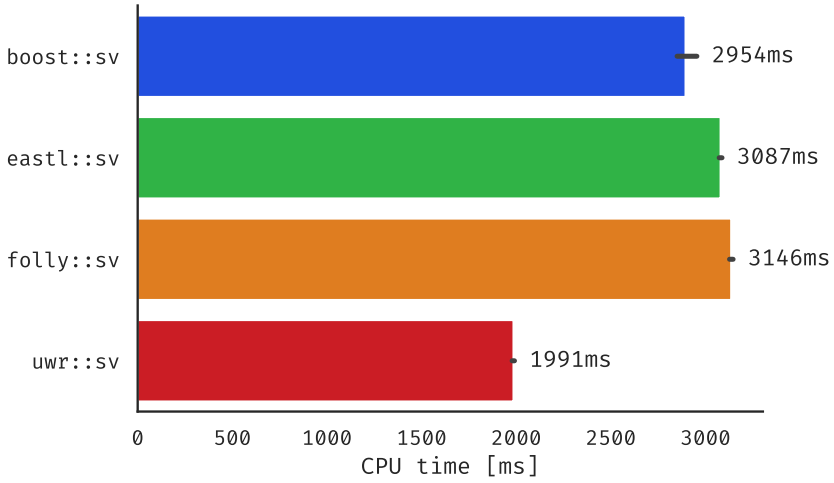

Figure 6.12: `all` benchmark for `int` value type, comparing `uwr::static_vector` with `push_back` optimization with alternative `static_vector` implementations.

ers no-worse performance than Boost alternative in all of the cases. However, it is still able to outperform `boost::static_vector` in some scenarios, offering more cache-friendly and more performant implementation. Together with `push_back` optimization turned on (which is turned on by default), it is the fastest existing option. Moreover with `uwr::static_vector` we are able to perform compile-time calculations by using `constexpr` keyword. Additionally, it is compliant with `C++20` standard, although, it is still compatible with `C++17`. It implements the whole `C++20` vector interface – contrary to the other alternatives.

6.5 `vector` results

We perform all three main types of benchmarks: `push-only`, `push-cons-dest` and `all`. Additionally, `push-only` benchmark is run in multiple setups. Every setup is characterized by the number of vectors in the environment. In this way, We want to investigate the performance and behavior of vectors under different circumstances – it is not uncommon to see a pattern of `vector` container usage to have a limited number of vectors where we only do `push_back` (and read, of course). Thus we want to check how the number of vectors affects the efficiency of a particular implementation. The setups for `push-only` benchmark are respectively for: 1, 10, 50, and 100 vectors in the environment.

6.5.1 *Trivial* types

We present benchmark results for two *trivial* types: `int` and `std::array<int, 10>`.

`int` type

`int` value type benchmarks are proposed to test vectors for storing small *trivial* type. The results for `push-only` variants are shown in Figure 6.13.

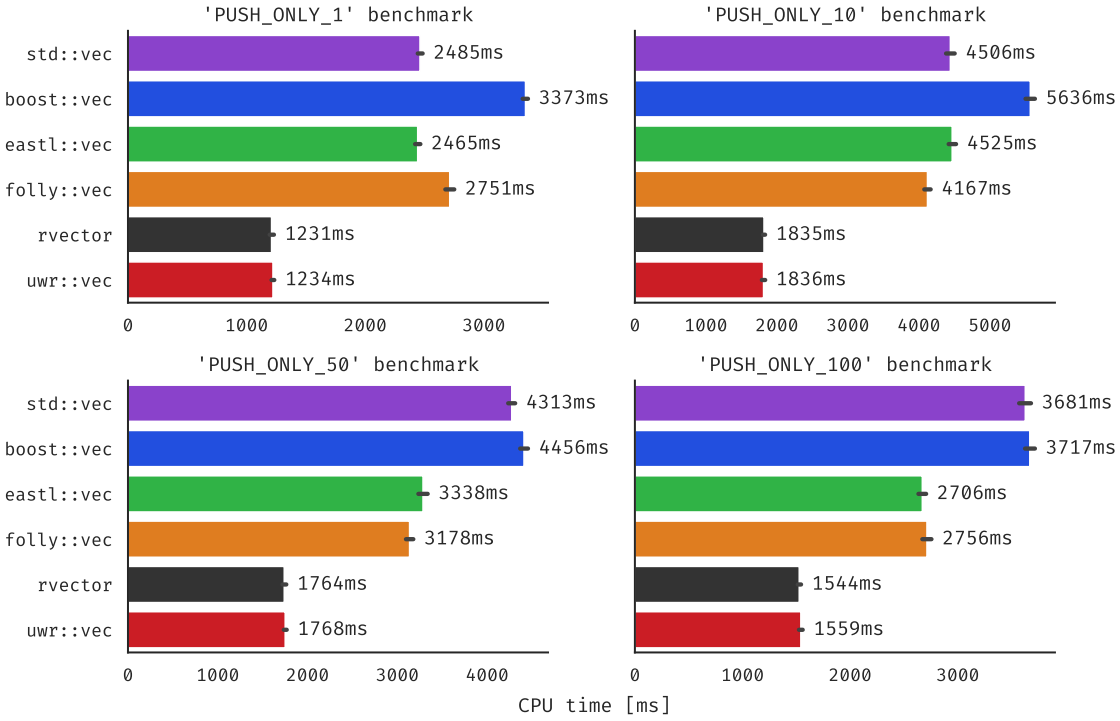

Figure 6.13: `push-only` benchmark in all variants for `int` value type, comparing `uwr::vector` with alternative `vector` implementations.

We can see that `uwr::vector` together with `rvector` perform much better than

all the other implementations in all variants. Note that for *trivial* types they do the same thing – they both use `mremap` call with `MREMAP_MAYMOVE` flag. This makes `mremap` call always succeed, and we do not have to use a custom policy for failures as for *non-trivial* types.

The situation is a little different for the two remaining benchmarks: `push-cons-dest` and `all` (Figure 6.14).

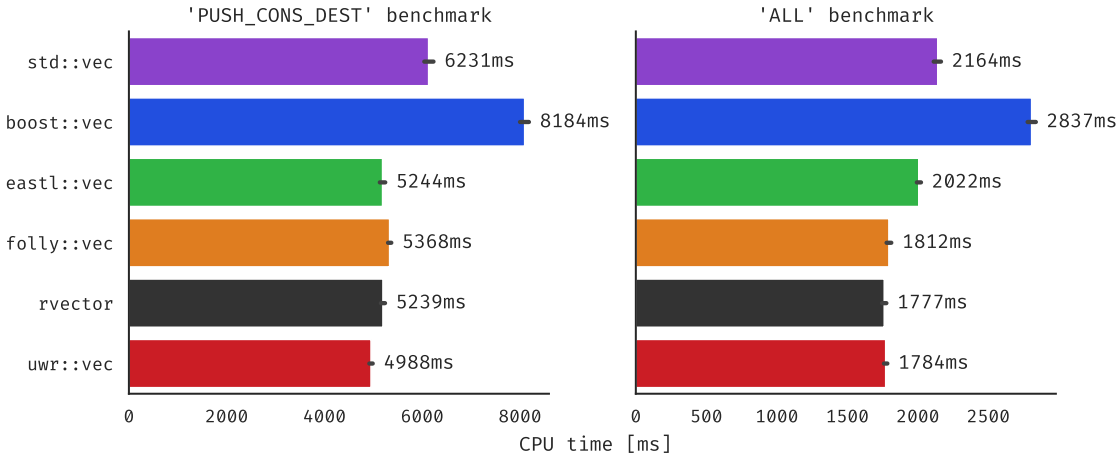

Figure 6.14: `push-cons-dest` and `all` benchmarks for `int` value type, comparing `uwr::vector` with alternative `vector` implementations.

Note that the number of vectors in the environment is larger compared to `push-only` benchmark – this means that the average vector size is smaller and the benefits from using `mremap` reallocation are reduced. Moreover, we perform a different set of actions for those benchmarks. They are not that focused on the vector growth compared to `push-only` alternatives, as we do `pop_back` and `erase` actions as well. Additionally, `int` is a small type so its reallocation is not that costly – we will see greater improvements for bigger types. Yet, `uwr::vector` and `rvector` still offer the best performance; just the differences are not as large as for `push-only` benchmarks.

`std::array` type

In order to test the vectors with a larger *trivial* type, we have used `std::array<int, 10>`. It is 10 times larger than `int`. Figure 6.15 and 6.16 show benchmark execution times for `push-only` variants and `push-cons-dest`, `all` respectively.

As we can see, `uwr::vector` together with `rvector` still performs far better than the other implementations for `push-only` benchmarks. Note that the differences for `push-cons-dest` and `all` benchmarks (Figure 6.16) are larger than for `int` type. It is because `mremap` gives higher benefits for larger *trivial* types, as the potential cost of moving them to another address is more expensive.

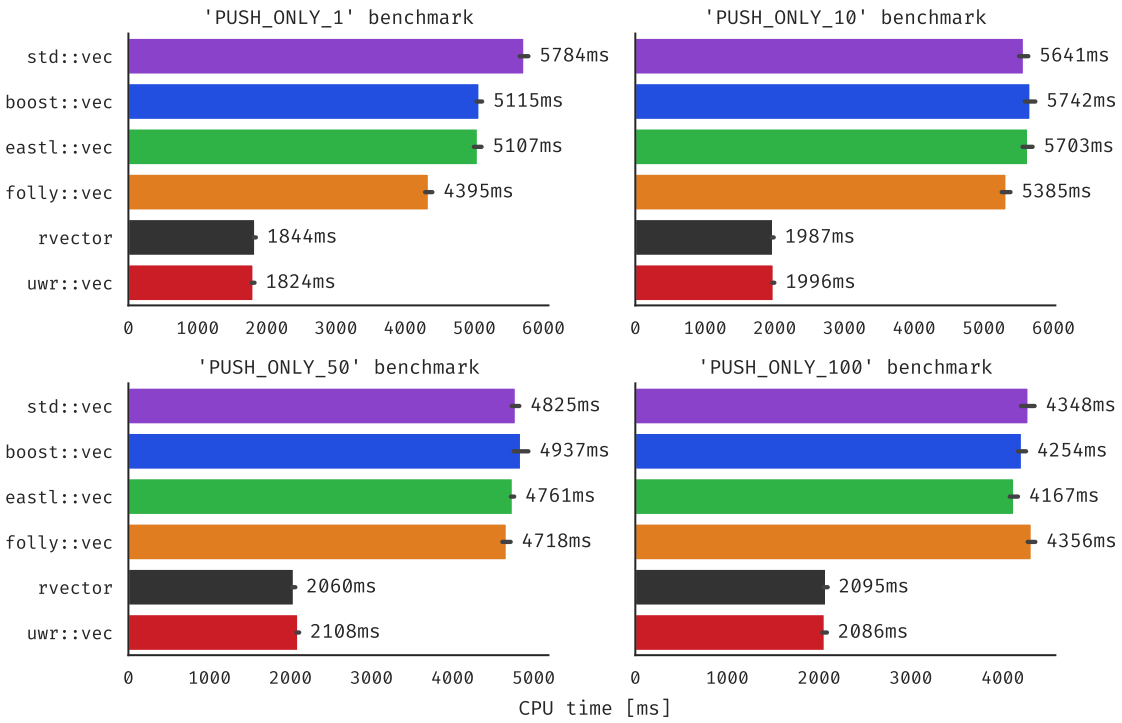

Figure 6.15: `push-only` benchmark in all variants for `std::array<int, 10>` value type, comparing `uwr::vector` with alternative `vector` implementations.

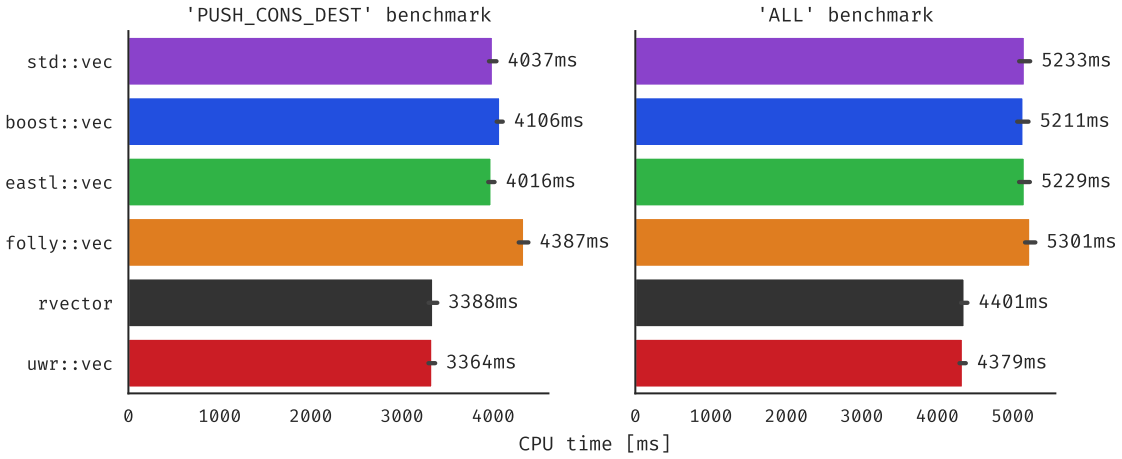

Figure 6.16: `push-cons-dest` and `all` benchmarks for `std::array<int, 10>` value type, comparing `uwr::vector` with alternative `vector` implementations.

Finally, to really emphasize the benefits of using `mremap` call for *trivial* types, we present Figure 6.17. It shows the iteration time of running `push-only` benchmark with only 1 vector in the environment.

Because we have only one vector, we can exactly see the moments when vectors grow – represented by spikes. Note that we do not see these spikes for both `rvector` and `uwr::vector`. The merit goes again to `mremap` call. Even though we perform a

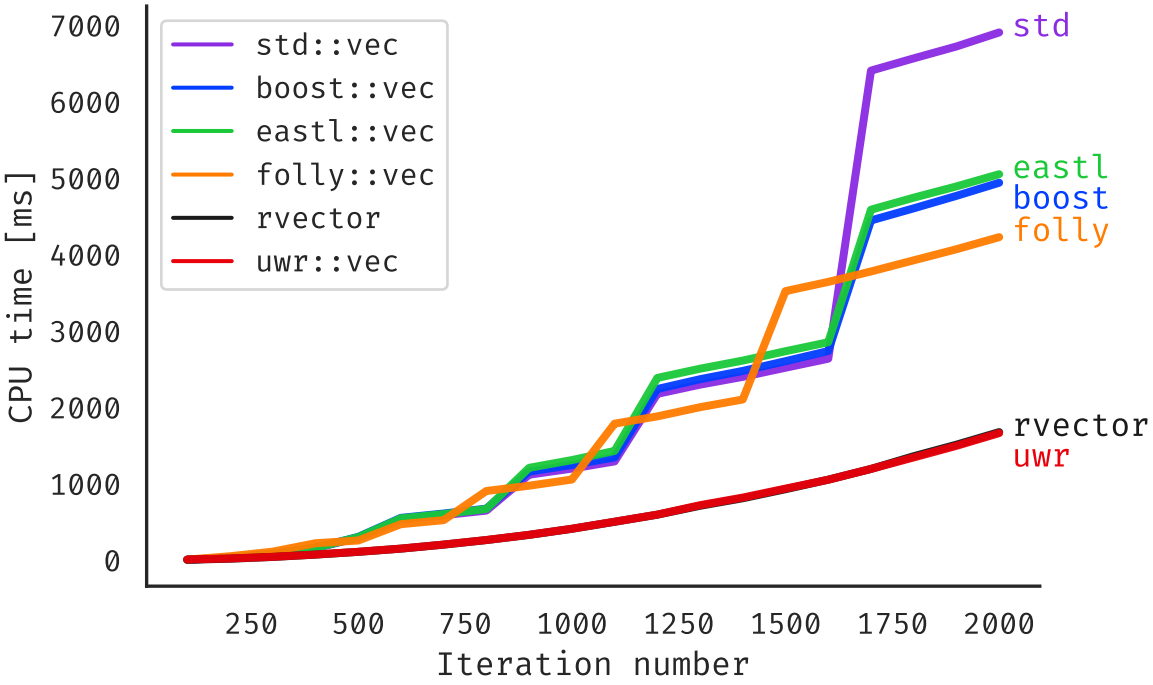

Figure 6.17: `push-only` benchmark with 1 vector in the environment for `std::array <int, 10>` value type, comparing iteration time of multiple `vector` implementations.

reallocation, we do not copy the actual data and it is visible on the plot.

6.5.2 *Non-trivial* types

We test vectors with the following *non-trivial* types: `std::string` and the custom type `test_type`. The implementation of `uwr::vector` for *non-trivial* types tries to increase the number of successful in-place reallocations. Recall that for *non-trivial* types we cannot use the `MREMAP_MAYMOVE` flag for `mremap` call, so rellocation in-place will sometimes fail.

`std::string` type

The results with `std::string` type are presented in Figures 6.18 and 6.19.

In Figure 6.18, we can see that `uwr::vector` has a clear advantage over all the other implementations. Note that the speed-up is bigger for a smaller number of vectors in the environment. This is partly caused by the fact that the larger number of vectors in the environment, the smaller is the average vector size: `uwr::vector` starts using `mremap` above a certain threshold, so then a smaller number of vectors qualify for remapping. Moving `std::string` via its move constructor is costly (compared to `memcpy`), so no wonder we see larger gains for bigger vectors as the in-place reallocation can take place and save a lot of time. Also, a larger number of

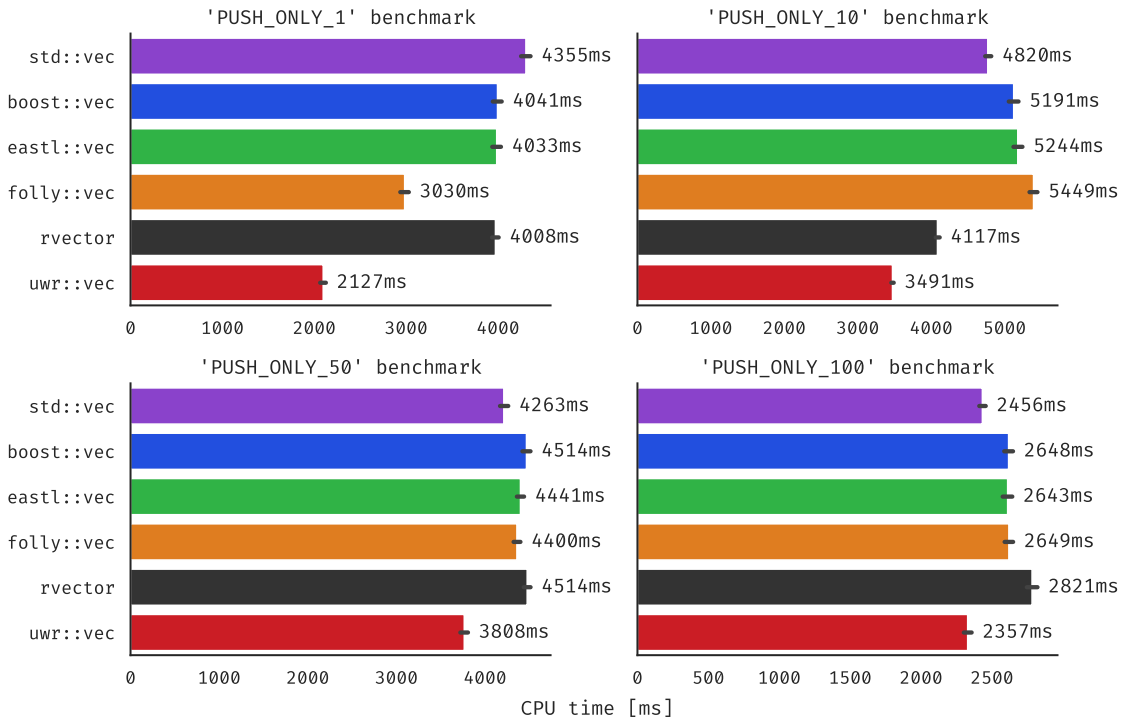

Figure 6.18: `push-only` benchmark in all variants for `std::string` value type, comparing `uwr::vector` with alternative `vector` implementations.

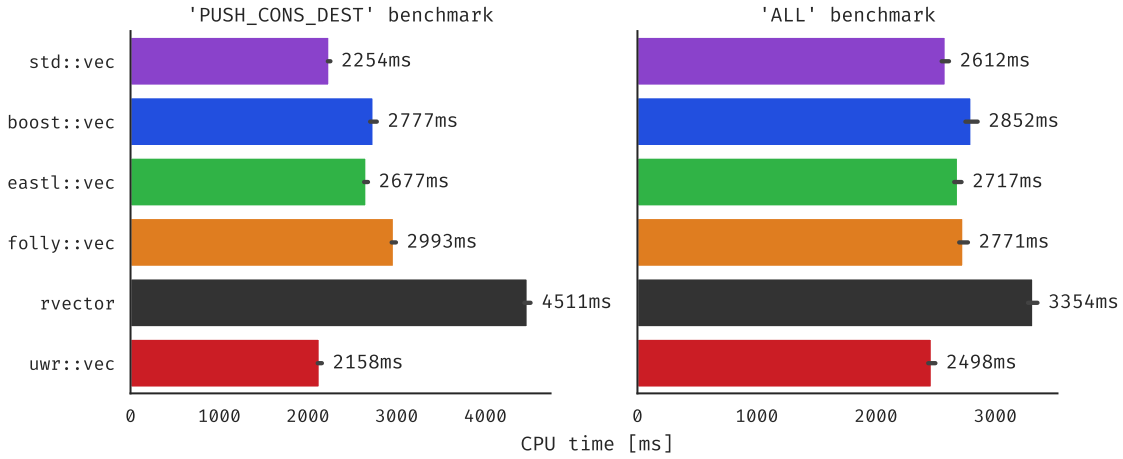

Figure 6.19: `push-cons-dest` and `all` benchmarks for `std::string` value type, comparing `uwr::vector` with alternative `vector` implementations.

vectors means increased memory fragmentation which can lower the `mremap` success rate. Indeed, for 1, 10, and 50 vectors in the environment, we have `mremap` success rate of approximately 50%, 45%, and 40%, respectively.

At the same time, we observed that the `mremap` success rate increased compared to that of `rvector`'s. Statistics for all the benchmark variants follow a similar pattern, e.g. for `PUSH_ONLY_10`, the success rate increased from 24% to 45%.

Figure 6.19 shows the results for `push-cons-dest` and `all` benchmarks. We have significantly more vectors in the environment compared to `push-only` cases. This measures `uwr::vector` performance also in a slightly different setup. Although we can see some speed-up, it is not as substantial as for `push-only` case. `uwr::vector` seems to be just slightly faster than `std::vector` in the average case. Its similarity as the number of vectors grows is due to the fact that more vectors use `malloc`–`free` pattern to perform a reallocation. On the other hand, `uwr::vector` is noticeably faster than `rvector`. `rvector` in those benchmarks has `mremap` reallocation rate not greater than 5%. It turns out it is not beneficial to use `mremap` call for *non-trivial* types at such an early stage and that using `malloc` is better. Additionally, having the constant growth factor equal to 2 seems to especially lower the success rate. It is often too big and induces many unnecessary non-in-place reallocations.

Finally, we present Figure 6.20. It shows iteration time of `push-only` benchmark with only one vector in the environment. In contrast to the same plot, but for `std::array` type, we can see the spikes caused by `uwr::vector` reallocation. Note that they are less frequent than for other `vectors`. We can see our in-place growth policy at work – intervals between successive reallocations are long, longer than for other `vectors`. This lets us save on the expensive reallocations and perform one only if it is necessary.

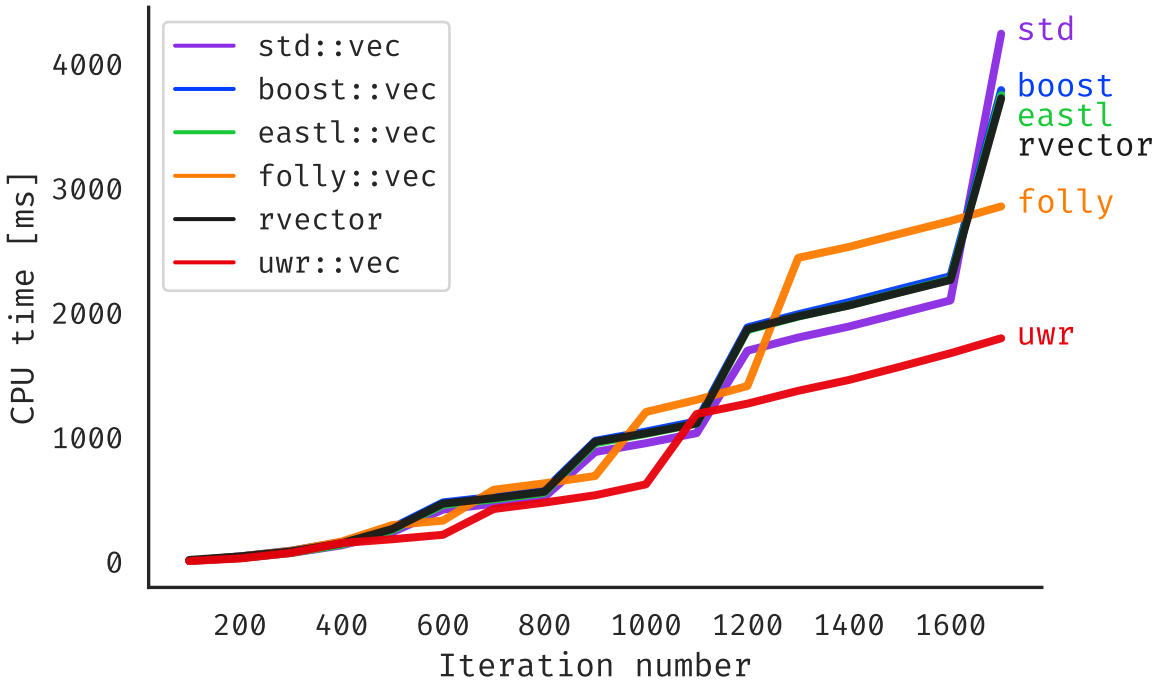

Figure 6.20: `push-only` benchmark with 1 vector in the environment for `std::string` value type, comparing iteration time of multiple `vector` implementations.

`test_type` type

`test_type` is a smaller type compared to `std::string` – it has only 8 bytes. Its benchmark results are presented in Figures 6.21 and 6.22.

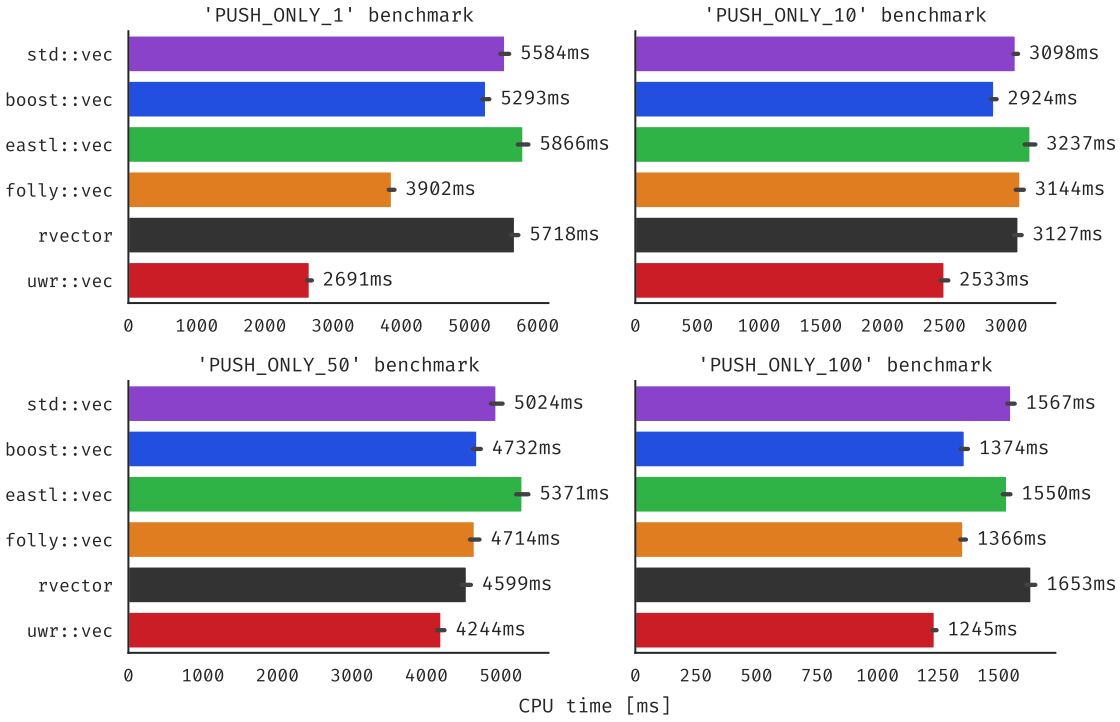

Figure 6.21: `push-only` benchmark in all variants for `test_type` value type, comparing `uwr::vector` with alternative `vector` implementations.

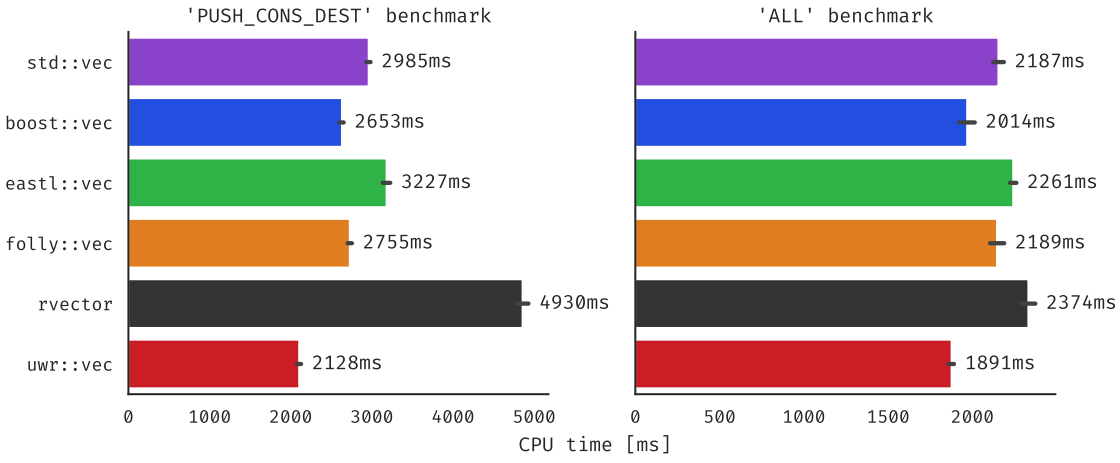

Figure 6.22: `push-cons-dest` and `all` benchmarks for `test_type` value type, comparing `uwr::vector` with alternative `vector` implementations.

We can see that the overall picture is quite similar to that for `std::string`. We have an advantage over all other implementations – a larger advantage for a lesser

number of vectors in the environment. Also note the similar speed-up compared to `rvector` in `push-cons-dest` and `all` benchmarks.

Lastly. let look at Figures 6.23 and 6.24. Again we can clearly see the benefits of using `mremap` by having a smaller number of reallocations – although we can see, that in 6.24, they are not as clear as in 6.23. The results are "averaged" over 10 vectors (i.e., this is the total time used by 10 `uwr::vector`s in the environment). Note, however, the influence of different *growth factor* policies. `folly::vector` has a factor of 1.5 for medium vector sizes. `uwr::vector` has a dynamic one due to maximal expansion policy during `mremap` failure. It makes the capacities to be out-of-sync with each other from the very beginning. This could lead to a situation when other vectors performed faster – e.g., after 2000th iteration (Figure 6.23) `uwr::vector`'s cumulative time was slightly larger than some implementations like `folly::vector`. If the benchmark would end there, we could have thought it is slower, when in fact, most of the time it has smaller time used. Exactly this situation happened to `folly::vector` and other vectors like `std::vector` and `boost::vector`. However, this is an inseparable part of every `vector` implementation – most of the time we do not know how much we should allocate next, so `vector` implementation has to "guess". Some guesses could be better at particular situations than others. Thus it is important to analyze the execution time during the whole simulation – this applies especially to a small number of vectors. We can see that this effect in Figure 6.24 has diminished, if not disappeared (at least for `uwr::vector`), as the benchmark time is affected by the combined performance of multiple vectors.

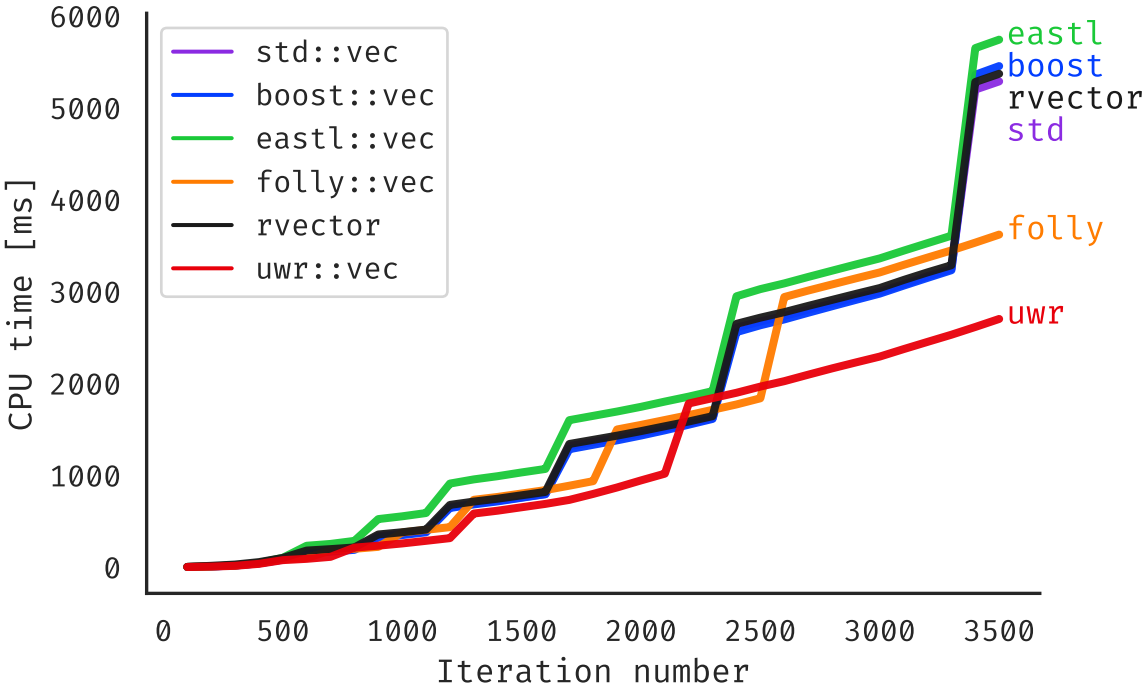

Figure 6.23: `push-only` benchmark with 1 vector in the environment for `test_type` value type, comparing iteration time of multiple `vector` implementations.

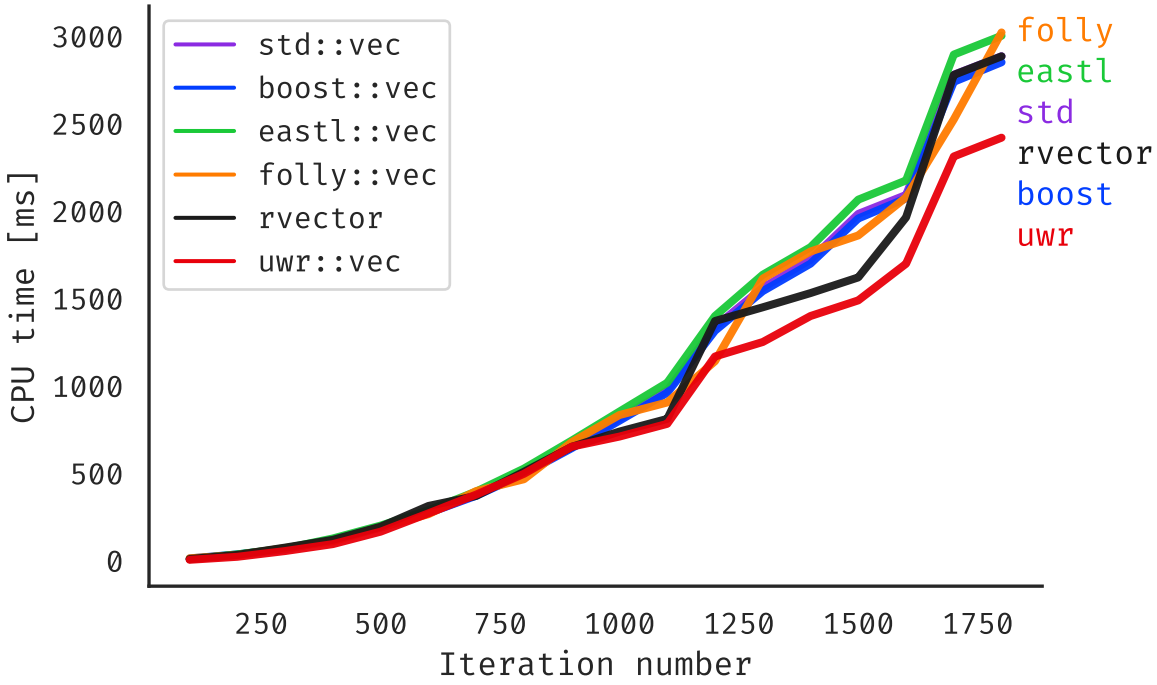

Figure 6.24: `push-only` benchmark with 10 vectors in the environment for `test_type` value type, comparing iteration time of multiple `vector` implementations.

6.6 Overriding triviality

The triviality of a type can be overridden using provided type traits – most importantly `uwr::mem::is_trivially_relocatable` trait, which enables using `mremap` with `MREMAP_MAYMOVE` flag.

We created a wrapper class around `test_type` – `T_test_type` – and defined it as a relocatable type:

```
struct T_test_type {
    test_type t;
};
```

Listing 6.3: T_test_type definition

```
namespace uwr::mem {

template<>
inline constexpr bool is_trivially_relocatable_v<T_test_type> = true;

}
```

Listing 6.4: Defining type as relocatable for `uwr::vector`.

`folly::vector` provides type traits to manually define a type as a *trivially-relocatable* as well. They were also used:

```
namespace folly {

template<>
struct IsRelocatable<T_test_type> : boost::true_type {};

}
```

Listing 6.5: Defining type as relocatable for `folly::vector`.

As we can see in Figures 6.25 and 6.26, considering this type as trivial provides substantial performance increase. It can be both observed in `folly::vector` and `uwr::vector` case.

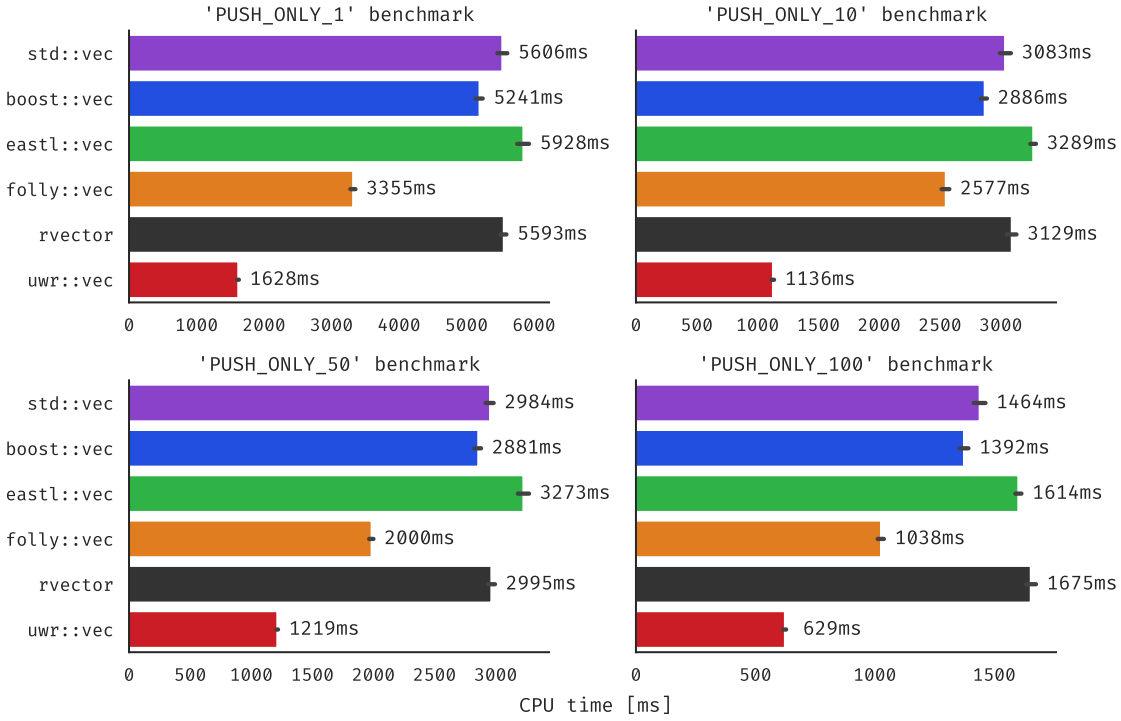

Figure 6.25: `push-only` benchmark in all variants for `T_test_type` value type, comparing `uwr::vector` with alternative `vector` implementations.

`uwr::vector` still performs better (at least no worse) than `folly::vector`. The reason for that is that, as we have already mentioned, we do not have to copy the data using `mremap` call. Looking at Figure 6.27, we can see spikes related to the vector's growth in the `folly::vector` case. They are globally smaller compared to other implementations, but they are still there. `uwr::vector` does not involve any spikes.

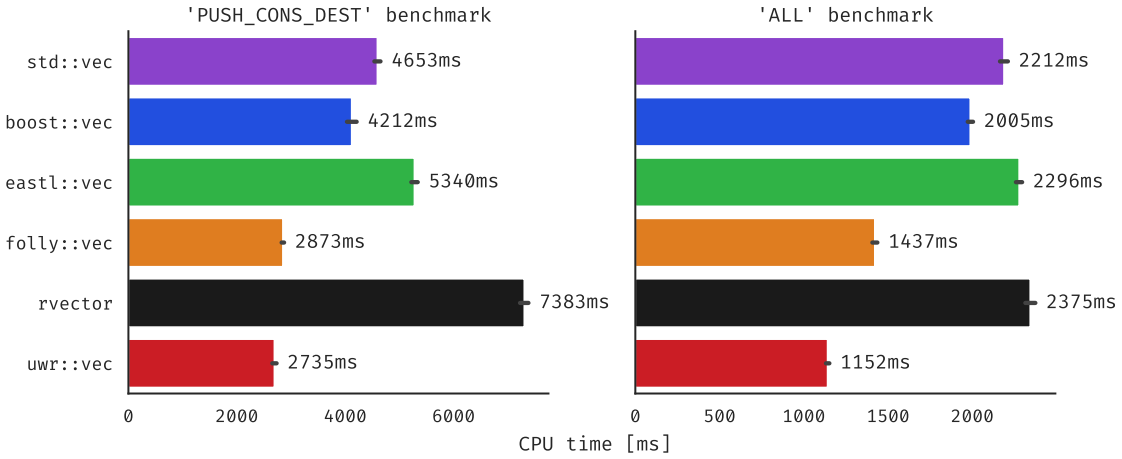

Figure 6.26: `push-cons-dest` and `all` benchmarks for `T_test_type` value type, comparing `uwr::vector` with alternative `vector` implementations.

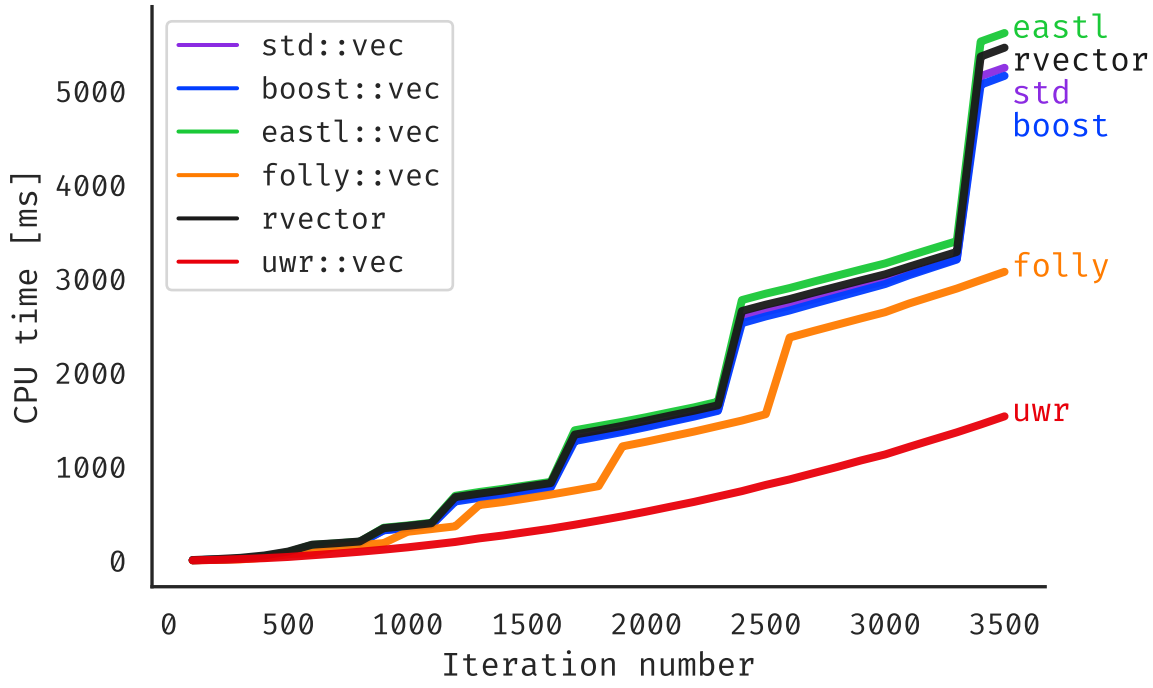

Figure 6.27: `push-only` benchmark with 1 vector in the environment for `T_test_type` value type, comparing iteration time of multiple `vector` implementations.

Chapter 7

Conclusions

`uwr::static_vector` and `uwr::vector` developed in this thesis provide great performance compared to the other alternatives. They both are cache-friendly and efficient container implementations. They are compliant with `C++20` standard and compatible with `C++17`. Moreover, they are very easy to integrate with a custom project as they are header-only and do not have any additional dependencies.

Additional `uwr::static_vector` features are as follows:

- All of its members are marked with `constexpr` keyword, hence it can be used in `constexpr` functions as a dynamically-resizeable container.
- It is size-frugal – small `static_vectors` can benefit from the size being minimal possible.
- It has `push_back/emplace_back` optimization compared to the Boost's alternative, which avoids code branching, thus additionally increases performance.

From the carried out results for both *trivial* and *non-trivial* types, we can conclude that `mremap` system call gives excellent opportunities to implement an efficient `vector`. The performance improvement are the greatest for *trivial* types, yet thanks to frugal reallocation policy for *non-trivial* types, we were able to save on some costly *non-trivial* reallocations and improve performance. Additional features and advantages of `uwr::vector` are as follows:

- It provides custom type traits that allow overriding the triviality of a type and significantly improve `vector` performance.
- It does not take any extra memory from the system – as it does not implement a custom allocator. It uses only `malloc` and `mmap` to allocate memory.

However, note that similarly to `rvector`, it is still provided only for Linux-based systems.

Possible future work includes the following aspects:

- `small_vector` implementation – this *vector* merges `vector` and `static_vector` functionality together, making it efficient for a limited number of elements, but still flexible to store larger quantities.
- `uwr::vector` is a Linux-only data structure. It should be ported to, e.g., the Windows operating system, utilizing existing in-place capabilities of allocators/system calls.

Bibliography

[1] Boost library source code. https://github.com/boostorg/boost. Accessed: 2021-09-04.

[2] boost::vector implementation. https://github.com/boostorg/container/blob/aa479c8eee0638536d62dda9eb03c4528f8fb35c/include/boost/container/vector.hpp. Accessed: 2021-09-04.

[3] Duff's device. https://en.wikipedia.org/wiki/Duff%27s_device. Accessed: 2021-09-04.

[4] EASTL source code. https://github.com/electronicarts/EASTL. Accessed: 2021-09-04.

[5] EASTL vector implementation. https://github.com/electronicarts/EASTL/blob/master/include/EASTL/vector.h. Accessed: 2021-09-04.

[6] Facebook's vector implementation. https://github.com/facebook/folly/blob/master/folly/FBVector.h. Accessed: 2021-09-04.

[7] Facebook's vector implementation description. https://github.com/facebook/folly/blob/master/folly/docs/FBVector.md. Accessed: 2021-09-04.

[8] Folly library source code. https://github.com/facebook/folly. Accessed: 2021-09-04.

[9] google-benchmark framework. https://github.com/google/benchmark. Accessed: 2021-09-04.

[10] gtest framework. https://github.com/google/googletest. Accessed: 2021-09-04.

[11] How to outperform std::vector in 1 simple step. https://codervil.blogspot.com/2012/11/how-to-outperform-stdvector-in-1-easy.html. Accessed: 2021-09-04.

[12] jemalloc implementation. https://github.com/jemalloc/jemalloc. Accessed: 2021-09-04.

[13] malloc reference. https://linux.die.net/man/3/malloc. Accessed: 2021-09-04.

[14] mremap Linux system call. https://man7.org/linux/man-pages/man2/mremap.2.html. Accessed: 2021-09-04.

[15] sbrk reference. https://linux.die.net/man/2/sbrk. Accessed: 2021-09-04.

[16] std::vector standard reference. https://en.cppreference.com/w/cpp/container/vector. Accessed: 2021-09-04.

[17] STL allocator design. https://web.archive.org/web/20150307090036/http://www.sgi.com/tech/stl/alloc.html. Accessed: 2021-09-04.

[18] Stefaan Poedts Dries Kimpe, Stefan Vandewalle. Evector: An efficient vector implementation – using virtual memory for improving memory. *Scientific Programming*, 2006. doi:10.1155/2006/690694.

[19] Arthur O. Dwyer. Trivialy relocatable type trait in C++. https://quuxplusone.github.io/blog/2018/07/18/announcing-trivially-relocatable/. Accessed: 2021-09-04.

[20] Wojciech Oziębły. A fast vector with mremap. https://github.com/Bixkog/rvector. Bachelor thesis. Accessed: 2021-09-04.